Cotas Asintóticas de Coste

Proyecto Fin de Máster
Programación y Tecnología Software

Diego Esteban Alonso Blas

Dirigido por: **Prof. Dª. Elvira Albert Albiol** y
Prof. Dª. Purificación Arenas Sánchez

Universidad
Complutense de
Madrid

Facultad de
Informática

Máster en Investigación en Informática
Curso 2008-2009

Septiembre de 2009

II

Autorización

D. Diego Esteban Alonso Blas, matriculado en el Máster de Investigación en Informática de la Facultad de Informática de la Universidad Complutense de Madrid (UCM), autoriza a dicha Universidad a difundir y utilizar con fines académicos, no comerciales y mencionando siempre a su autor el presente Trabajo Fin de Máster: ***Cotas Asintóticas de Coste***, realizado durante el curso académico 2008-2009 bajo la dirección de las Profesoras Dª. Elvira María Albert Albiol y Dª. Purificación Arenas Sánchez, en el Departamento de Sistemas Informáticos y Computación. Autoriza al tiempo a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

En la Facultad de Informática de la Universidad Complutense de Madrid, Calle del Profesor García Santesmases, en Septiembre de dos mil nueve.

Fdo. Diego Esteban Alonso Blas

Resumen

Resumen

La complejidad asintótica de un programa describe la escalabilidad de su coste de ejecución. Observa su comportamiento cuando procesa datos arbitrariamente grandes y considera semejantes dos programas cuyos costes crecen de manera similar. Existen algunos analizadores automáticos de coste que, cuando se emplean para procesar programas reales, generan funciones no asintóticas de coste demasiado complejas para su uso práctico. En este trabajo **desarrollamos e implementamos un algoritmo capaz de simplificar esas funciones a su forma asintótica**, más compacta y manejable. **Hemos integrado dicho algoritmo en un resolutor de ecuaciones**, capaz de generar directamente funciones asintóticas sin necesidad de calcular las no asintóticas; lo que en nuestras pruebas experimentales ha resultado ser más eficaz. Ello permite mejorar la escalabilidad de los analizadores de coste, un aspecto crucial para su uso industrial.

Abstract

Asymptotic complexity analysis is used for describing programs' excution cost scalability, by observing its behaviour when processing large input data and considering as equivalent two programs whose costs grow at the same rate. There are *non-asymptotic* cost analyzers which, when used over realistic programs, provide complex non-asymptotic cost functions that can't be used in some practical applications. That's why **we have developed an algorithm that simplifies a cost function to its asymptotic form**, more compact and manageable. **We have integrated that algorithm into a cost analyzer**, and now it generates asymptotic functions without having to firstly compute their non-asymptotic counterparts, a process that our experiments show to be more efficient. This allows us to improve cost analyzers' scalability, a concerning issue for its application in software industry.

Palabras Clave

- Análisis de Coste
- Análisis Estático de Programas
- Análisis Automático de Complejidad
- Cotas Superiores en Forma Cerrada
- Interpretación Abstracta
- Código con Certificados
- Lenguajes de Programación
- Código Byte de Java
- Órdenes de Complejidad Asintótica

Keywords

- Cost Analysis
- Static Program Analysis
- Automatic Complexity Analysis
- Closed-Form Upper Bounds
- Abstract Interpretation
- Proof Carrying Code
- Programming Languages
- Java Bytecode
- Asymptotic Complexity Order

Material y métodos

Este Trabajo de Fin de Máster es una recopilación de los conceptos, algoritmos y técnicas presentadas en varios artículos científicos y, de manera primordial, en:

1. ***" Closed-Form Upper-Bounds in Static Cost Analysis "*** (Cotas Superiores en Forma Cerrada para Análisis de Coste) escrito por Albert, Arenas, Genaim y Puebla en 2008 [10] y

2. ***" Asymptotic Resource Usage Bounds "*** (Cotas Asintóticas del Consumo de Recursos) escrito por Albert, Arenas, Genaim, Puebla y el autor de este trabajo en 2009 [2].

así como [5, 8, 3], que recogen la investigación efectuada por los miembros de este grupo de investigación, compuesto por investigadores de las Facultades de Informática de las universidades Complutense y Politécnica de Madrid.

En los Capítulos 2 y 3 repasamos los conceptos y métodos del primer documento, *(Closed-Form Upper-Bounds in Static Cost Analysis)*. En él se desarrolla un algoritmo para obtener resultados en forma cerrada de relaciones de coste. Este método ha resuelto de manera satisfactoria un problema que llevaba abierto en este campo desde los años '70.

Las aportaciones del segundo documento *(Asymptotic Resource Usage Bounds)* se exponen en el Capítulo 4. Su mayor innovación es el algoritmo que simplifica a "forma normal asintótica"una amplia clase de funciones de coste. Esto ya sirve para reutilizar la investigación y los artefactos en resolución de ecuaciones, pero su mejor aplicación es para construir un resolutor asintótico más rápido que los no asintóticos.

El presente Trabajo revisa los conceptos y algoritmos presentados en esos artículos y les da una descripción más detallada, aprovechando la amplitud que no hay en publicaciones científicas. Algunos de ellos, sobre todo los del

Capítulo 2, se presentan enfocados y extendidos para que sirvan de base de trabajos futuros.

Para ayudar a entender las ideas expuestas y facilitar un uso didáctico, cada concepto se complementa con ejemplos ilustrativos, que van de simples "juguetes"a ejemplos más complejos obtenidos en el análisis de algunos programas que incluyen características de software real.

Agradecimientos

Agradecimientos personales

A las directoras de este trabajo, las profesoras Elvira Albert y Purificación Arenas, por haberme dado la oportunidad de trabajar e investigar en este fértil campo de investigación, por haber confiado en mis capacidades y por su dedicación en la corrección de mis errores.

Asimismo, a los profesores Germán Puebla, Samir Genaim, Damiano Zanardini y Jesús Correas, y también a los doctores Diana Ramírez, Israel Herraiz y Miguel Gómez Zamalloa, por desarrollar COSTA. Sin su innovadora obra este proyecto no hubiera sido posible.

Y a mis padres por su apoyo y ánimo en todo momento.

Agradecimientos institucionales

Este Trabajo ha sido parcialmente financiado por:

- el programa de **Tecnologías de la Sociedad de la Información** *(Information Society Technologies)* de la Comisión Europea, de Tecnologías Emergentes y Futuras *(Future and Emerging Technologies)* bajo el proyecto marco IST-231620 HATS *(Highly Adaptable and Trustworthy Software using Formal Methods)*,
- por el Ministerio de Educación y Ciencia bajo los proyectos TIN-2008-05624 **DOVES (Development Of Verifiable and Efficient Software)** y HI2008-0153 (Acción Integrada),
- por el proyecto S-0505/TIC/0407 PROMESAS **(PROgrama en MÉtodos para el Desarrollo de Software Fiable, de Alta Calidad y Seguro)** de la Comunidad Autónoma de Madrid,
- y por el UCM-BSCH-GR58/08-910502 (GPD-UCM).

Índice general

Capítulo 1

Análisis de Coste

Este Capítulo presenta el campo del Análisis Estático de Coste, para orientar al lector sobre los objetivos, conceptos esenciales y métodos de este campo y, con ellos, de este trabajo.

1.1. Costes y Análisis

Una máquina se construye para ejecutar una tarea con menos esfuerzo, y su valor está en la relación entre su funcionalidad y su coste. Dado que un programa es una máquina, para evaluarlo se debe conocer el coste de ejecutar con él una tarea.

1.1.1. Costes

El gráfico de la Figura 1.1 representa esquemáticamente la ejecución de un programa en un computador. En ésta los datos de entrada se transforman en resultados de salida, siendo la relación entre unos y otros la semántica del programa. Para ello el computador gasta unos recursos y genera unos costes.

Definición 1 (Coste de Ejecución)**.** Un **Coste de Ejecución *(Execution Cost, EC)*** es una magnitud que mide el **consumo de recursos** *(resource consumption)* derivado de la ejecución de un programa. □

Estos son algunos de los costes más interesantes en Informática:

- **Instrucciones** máquina ejecutadas, pudiendo distinguir entre operaciones enteras o reales, saltos, accesos a memoria, etc.
- **Memoria** que ocupan los objetos de datos ubicados en la pila o montículo, y número de accesos a cada uno.

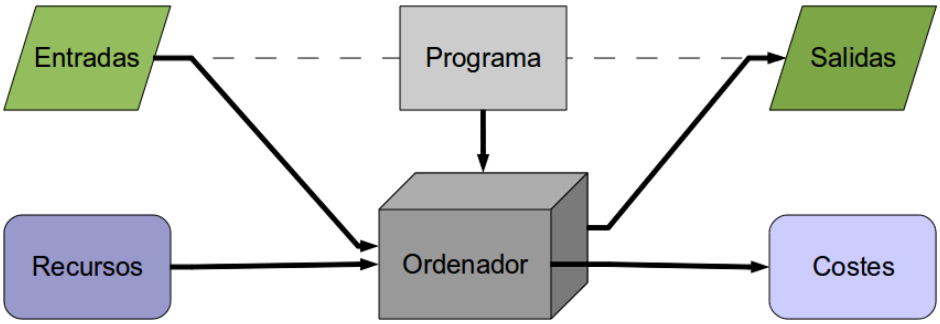

Figura 1.1: Esquema de una Ejecución.

- Datos o mensajes enviados por una red de telecomunicación, lo que a veces implica eventos facturables como los SMS en en teléfono móvil.
- Costes para requisitos: para asegurar la confidencialidad e integridad de un dato interesa controlar qué procesos acceden a él y cuánto. O también, para medir el acoplamiento con una biblioteca externa (o sistema) se mide el número de llamadas a sus funciones.

1.1.2. Análisis de Coste

Buscamos un método capaz de estudiar costes de ejecución. Éste debe decirnos (1) cómo **comparamos** el coste de varios programas, (2) con qué unidad se **mide**, (3) en qué **circunstancias** valen nuestros resultados y (4) qué mecanismos tenemos para **verificarlos**. Inspirados en otras ramas de la ciencia e ingeniería, se han desarrollado dos grandes enfoques de estudio.

En el enfoque dinámico-experimental: se mide el consumo de varias ejecuciones, y se analiza cómo varían las mediciones al cambiar de computador, de programa o de datos de entrada. Un ejemplo son las suites de *benchmarks* SPEC, para comparar el rendimiento de procesadores.

En el enfoque estático: se usa la interpretación abstracta [23] para, del programa original, obtener la información correcta más precisa posible sobre el coste de ejecutarlo en cualquier entrada. Se basa en estos postulados:

Abstraemos la computadora al suponer que el coste de ejecutar el programa $P \in \mathbb{P}$ sobre unos datos de entrada $\overline{x} \in \overline{\mathbb{D}}$, escrito $P(\overline{x})$, solo depende del programa y de esos datos. La Figura 1.2 (Página 3) muestra el esquema de la Figura 1.1 (Página 2) al quitar el ordenador.

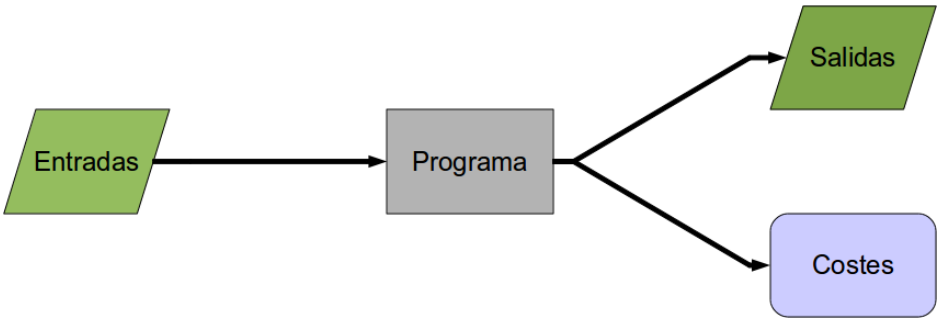

Figura 1.2: Ejecución abstracta, sin computadora.

Abstraemos el programa en una **función de coste** *(cost function)* $f_P : \overline{\mathbb{D}} \mapsto \mathbb{R}^+$, que para cada dato $\overline{x}$ devuelve el coste de $P(\overline{x})$. Este paso se muestra de manera gráfica en la Figura 1.3.

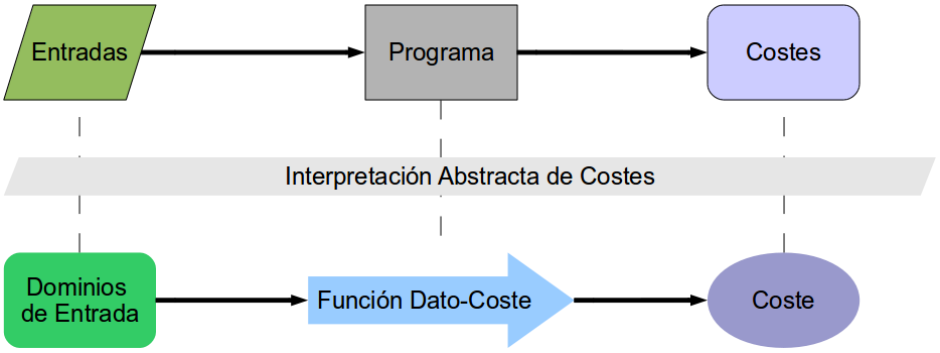

Figura 1.3: Interpretación del programa con un modelo de coste.

Hacer esto manualmente es difícil, propenso a error y poco escalable. Por ello cobra importancia el **análisis de coste** *(cost analysis)*, también conocido como **de consumo de recursos** *(resource consumption analysis)*, por Wegbreit [57], Debray [27] Gulwani et al. [29] y Sands [52]; o como análisis **de complejidad** *(complexity)*; que estudia cómo implementar analizadores de coste.

1.1.3. Tamaños y Cotas

En muchos programas la función dato-coste es muy compleja porque los dominios de los los datos de entrada $\mathbb{D}_i$ son tipos estructurados como listas, matrices, tablas clave-valor, grafos, etc. La abstracción de tamaños simplifica el análisis de esos programas empleando las funciones de tamaño y la relación de tamaño-costes (Figura 1.4, Página4).

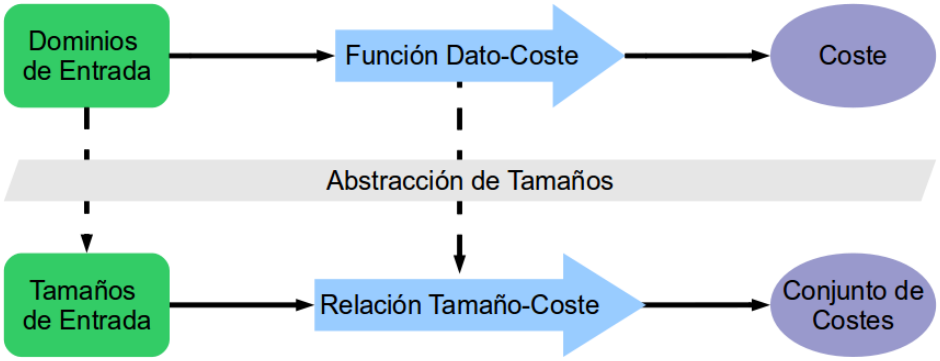

Figura 1.4: Abstracción de tamaño en análisis de costes.

Definición 2 (Medida de tamaño)**.** Sea $\mathbb{D}$ un dominio de datos. Una **medida de tamaño** *(size measure)* es una función $s : \mathbb{D} \mapsto \mathbb{N}$. □

Algunas medidas usuales de tamaño son el valor de un número, la longitud de un array, el número de elementos en una colección, la longitud de camino más largo de una estructura de datos anidada (Spoto et al. [54]), etc.

Definición 3 (Relación tamaño-coste.)**.** Sea $\overline{\mathbb{D}}$ una serie de m dominios de datos, cada uno con una medida de tamaño s_i, y sea $f_P : \overline{\mathbb{D}} \mapsto \mathbb{R}^+$ una función de dato-coste. La **relación tamaño-coste** *(size-cost relation)* de f_P es $f'_P : \mathbb{N}^n \mapsto \mathcal{P}(\mathbb{R}^+)$ definida como

$$f'_P(\overline{n}) = \{r \in \mathbb{R}^+ | \exists \overline{x} \in \mathbb{D}^n : r = f_P(\overline{x}) \wedge \overline{n} = (s_1(x_1), ..., s_m(x_m))\} \quad (1.1)$$

que para $\overline{n}$ da los costes de vectores de entrada de tamaños $\overline{n}$. □

Sobre f'_P se definen las funciones de coste máximo y mínimo

$$max_P = max \circ f'_P \qquad min_P = min \circ f'_P \quad (1.2)$$

La abstracción de tamaños introduce imprecisión en el análisis porque cada valor del tamaño puede representar muchos datos. Para tratar esa imprecisión es necesario aproximar la relación tamaño-coste con funciones que describan todo el conjunto de respuestas. Las más comunes son:

- El estudio de **caso pésimo** *(worst-case)* busca funciones ub_P de **cota superior** *(upper bound)*: $ub(\overline{n}) \geq max_P(\overline{n})$.
- Para encontrar una **cota inferior** *(lower bound)* $lb(\overline{n}) \leq min_P(\overline{n})$ se emplea el estudio de **caso óptimo** *(best-case)*. Una función que es a la vez cota superior e inferior da el coste exacto.
- En los estudios de **caso medio** *(average case)* se buscan estimaciones estadísticas-probabilísticas, como la media de los costes reales.

1.2. Aplicaciones del análisis de coste

En esta sección presentamos algunas aplicaciones que el análisis automático de coste tiene en la industria informática.

1.2.1. Desarrollo de programas

El usuario de una aplicación quiere que ésta cumpla unos requisitos de eficiencia, seguridad, robustez, etc. En algunas metodologías de desarrollo se asume que, como estos requisitos describen propiedades emergentes del sistema, no se pueden verificar antes de completarlo y por ello separan la gestión de estos requisitos del proceso de desarrollo de software. El análisis de costes permite reintegrar esa gestión en dicho proceso.

En la gestión de requisitos, los requerimientos no funcionales pueden formalizarse con aserciones de coste. Éstas proporcionan una base para decidir la viabilidad al principio del proyecto, y una vez está completado la Verificación consiste en comprobar algorítmicamente dichas aserciones.

En el diseño arquitectónico se deben indicar qué costes se esperan en cada subsistema. Si esto se hace con aserciones de coste; sería posible combinar sobre *el papel* los costes de cada módulo, para decidir algorítmicamente si esa arquitectura garantiza que la aplicación cumplirá sus requisitos.

Al reutilizar componentes genéricos, se deben evaluar cómo satisface nuestros requisitos cada uno de los componentes alternativos que implementan una funcionalidad, y escoger cuál se integra en la aplicación. Si esos requisitos se dan como aserciones de coste, el análisis automático puede realizar esa evaluación e incluso escoger cuál se emplea.

En la síntesis y optimización automática de programas se estudian técnicas que mejoran un programa preservando su semántica, como la evaluación parcial (Craig y Leuschel [25], Puebla y Ochoa [49]). Éstas siguen un proceso indeterminista: generan varios programas mejorados entre los cuales se debe elegir el óptimo. Esta elección puede realizarla automáticamente un analizador de coste.

La depuración del programa es más sencilla y rápida si un analizador de coste localiza los fallos de programación. Para ello solo tiene que comprobar algunas aserciones de coste que especifique el programador en el código. Esta

aplicación ya la mencionó Ben Wegbreit en [57]. Para depurar programas en PROLOG, esta funcionalidad fue implementada en el preprocesador CIAOPP por M. Hermenegildo, G.Puebla et al. en [31]. En los próximos meses estará disponible un *plugin* para ECLIPSE que permitirá depurar aserciones en un programa JAVA, desarrollado por Román-Díez et al. en [4].

La escalabilidad es relevante para el mantenimiento a largo plazo. Según crecen las prestaciones del hardware y las necesidades de los usuarios, importa saber cómo mejora el funcionamiento de un programa al instalarlo en una nueva máquina o al emplearlo con datos más grandes. Una buena manera de saberlo es analizando las relaciones de tamaño-coste.

1.2.2. Sistemas operativos

Un Sistema Operativo (S.O) debe repartir los recursos del computador entre los procesos. Esto puede hacerse mejor si se conocen los costes de cada proceso.

La gestión de memoria debe minimizar los fallos en la jerarquía de memoria, lo que se logra asignando a los niveles superiores aquellas porciones a las que más se va a acceder en el futuro reemplazando las que no son necesarias. Si el SO sabe cuánto ocupan los datos de un proceso y cuántas veces se va a acceder a ellos entonces realizará una planificación más precisa.

En los sistemas paralelos, como los procesadores *multicore*, los *clusters* y el *grid computing*, el SO debe realizar una división, localización y planificación de los procesos paralelos entre los nodos. Estas gestiones reciben el nombre de equilibrio de carga por lo que recibe el nombre de equilibrio de carga por Debray [27] y por Hermenegildo, Albert, Puebla et al. [30]. Se puede usar la información de costes para hacer un equilibrado más preciso, que asigne a todos los procesadores el mismo número de instrucciones máquina. También se puede reducir la sobrecarga de comunicaciones si se ubican los procesos más acoplados en nodos cercanos.

En un sistema de tiempo real cada tarea debe completarse antes de un plazo límite o *(deadline)*. Si el SO sabe cuántas instrucciones puede ejecutar como máximo cada tarea, puede planificar su ejecución a fin de que un número mayor de tareas (o las más prioritarias) se completen a tiempo.

1.2.3. Distribución y Certificación de Código.

Existen ordenadores cuyas prestaciones son limitadas, como los reproductores multimedia o los terminales móviles; o que operan en un entorno de alto riesgo, como son los sistemas de control industrial. Sea por el riesgo o sea por el precio, en ambos casos es muy importante garantizar que las aplicaciones instaladas no provocarán fallos de funcionamiento *(failures)*.

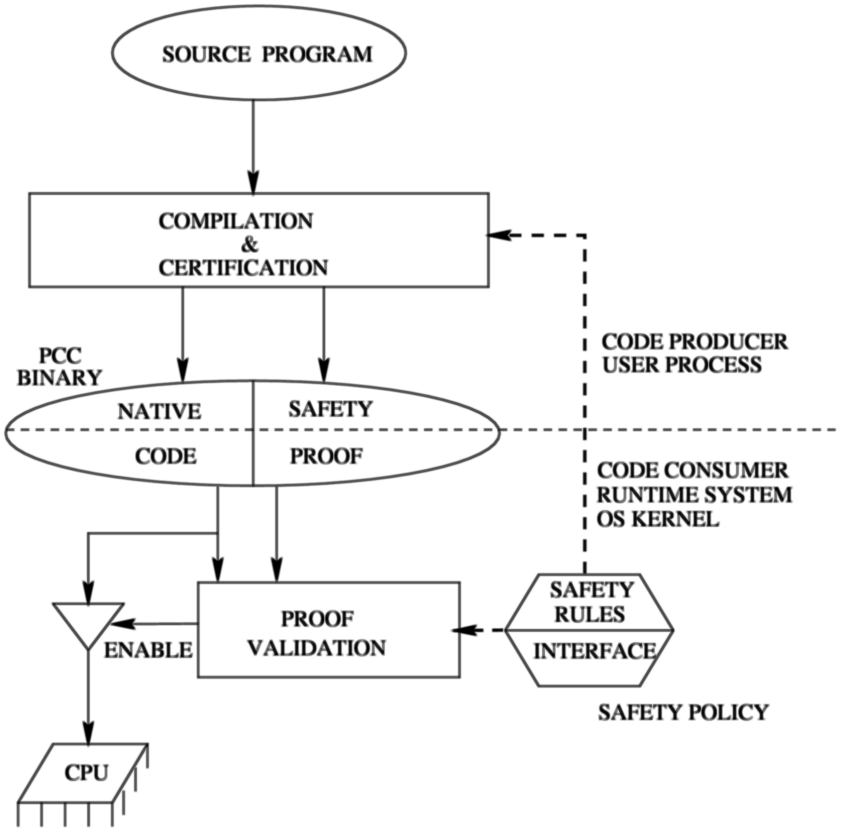

Figura 1.5: Esquema de George Necula del Código Certificado[46].

En ambos entornos la cuestión es cómo garantizar el correcto funcionamiento de un programa de manera que el cliente pueda verificar esa garantía. Para resolverla George Necula propuso en 1997 el modelo de **código certificado** *(Proof Carrying Code PCC)*[46]. Como muestra la Figura 1.5, al programa se anexa un certificado de propiedades de seguridad. Un comprobador *(checker)* verifica esas propiedades y si éstas cumplen la política de seguridad del sistema, y en tal caso acepta la aplicación. Los trabajos previos en PCC se restringen a cotas lineales (Aspinall et al. [11], Crary [26] y Hoffman [33]) y técnicas semiautomáticas de Necula et al. [20].

1.3. Arquitectura de Wegbreit

La investigación del análisis automático de coste basado en construir expresiones funcionales no recursivas fue iniciada por Ben Wegbreit en 1975 en [57], artículo en que propuso una arquitectura de analizadores:

Figura 1.6: Descripción de las fases de la Arquitectura de Wegbreit

1.3.1. Fase 1: Obtención del CRs

Las entradas de esta fase son el código del programa, en un lenguaje de programación (fuente), y un modelo de costes para ese lenguaje. El resultado es una **relación de coste** *(cost relation, CR)*, que modela el coste de ejecución del programa en función de sus datos de entrada.

Definición 4 (Modelo de Coste)**.** Un **modelo de coste** *(cost model)* de un lenguaje L es una especificación semántica de éste, que dice como traducir un programa escrito en L a una CR. □

Existen muchas implementaciones de esta fase, diseñadas para lenguajes diferentes. La Figura 1.7 muestra la arquitectura de COSTA, un analizador de programas en *Java Bytecode* con diversos modelos de coste, desarrollado por Albert, Arenas, Puebla, Genaim y Zanardini [7].

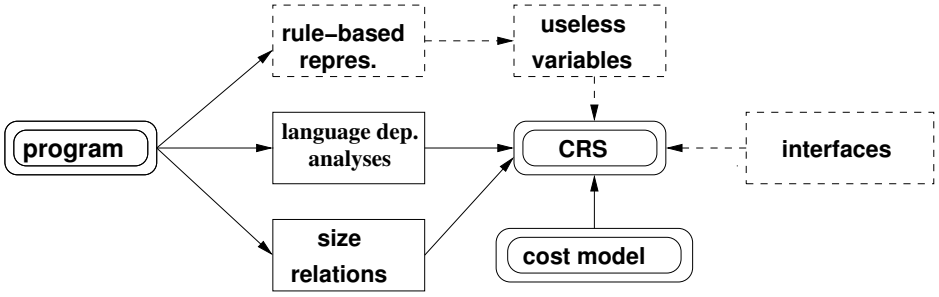

Figura 1.7: Arquitectura de COSTA [7], un Analizador de la Primera Fase

Cabe destacar los siguientes procesos de COSTA:

1. Se construye el **grafo de control de flujo** *(control flow graph CFG)* del programa. Sobre el CFG se estudian cuáles son las construcciones iterativas (bucles) del programa fuente, las cuales determinan la estructura recursiva del *CRS*,

2. Se procesa el programa con el modelo de costes, para obtener un **programa con cuenta de pasos** *(step-counting program)*, sustituyendo las instrucciones del lenguaje por los valores de sus costes.

3. Se transforma el programa mediante interpretación abstracta (Cousot [23]) para aproximar cómo cambian las variables durante la ejecución del programa. En esencia, esto consiste en abstraer esas variables con medidas de tamaño (Subsección 1.1.3) e inferir **restricciones de tamaño** *(size constraints)*, también llamadas **relaciones de tamaño** *(size relations)*, entre las variables del programa en distintos puntos del programa. Tal abstracción se conoce como **abstracción de tamaño** *(size abstraction)* y a este análisis como **análisis de tamaño** *(size analysis)*.

4. Generamos las relaciones de coste al combinar los costes locales obtenidos en el paso 2 con las abstracciones de tamaño del paso 3. Una vez obtenidas, se realizan procesos para simplificarlas y eliminar información redundante. Por ejemplo, se suprimen las variables que no afectan al coste, usando técnicas de *slicing* como muestran Zanardini et al. [6].

No daremos más detalles técnicos de esta fase porque excede el ámbito de este trabajo. Sí vamos a comentar la corrección y completitud de sus resultados. Cuando COSTA analiza un programa el resultado es una CR que recoge completamente el coste de las posibles ejecuciones del programa, pero que no siempre es correcta pues puede admitir un valor que no corresponde a ninguna ejecución, como se ve en el Ejemplo 3 (Página 12).

Ejemplos de Análisis de Coste

Vamos a describir el proceso de análisis de coste con un ejemplo, mostrando algunos resultados intermedios que COSTA genera.

Ejemplo 1 (Análisis del número de instrucciones de Lista.delete)**.** Vamos a analizar el método delete de la clase Lista (Figuras 1.8 y 1.9). Aunque COSTA analiza Java Bytecode [7], mostramos el código Java porque es más legible.

El método Lista.delete tiene cuatro parámetros: lis es una Lista sin repeticiones, piv es un valor entero, va es un Vector que solo contiene enteros

```
// Vector: array con posiciones ocupadas
public class Vector{
    int[] datos;     // Valores en el Vector
    int   size;      // Numero de posiciones ocupadas
    // Invariante: 0 <= size < datos.length

    public void elimina(int val){
        int i=0;
        while (i < size && datos[i]<val) {
            i++; //Cuerpo: Ecuacion (7)
        };   // Salida: ecuaciones (5),(6)
        size --;
        for (int j=i; j<size; j++){
            datos[j]=datos[j+1]; // Cuerpo: Ec. (9)
        } // Salida de bucle: Ec. (9)
    }

    /** Suma los elementos del Vector      */
    public Integer suma(){
        int sum = 0;
        for(int i=0;i<size; i++){
            sum += datos[i];
        }
        return new Integer(sum);
    }
}
```

Figura 1.8: Definición de clase Vector.

menores que piv y vb es un Vector que solo contiene enteros mayores o iguales que piv. La funcionalidad de Lista.delete es eliminar de los Vectores va, vb los valores que aparecen en lis.

La Figura 1.10 muestra el CFG del programa, separándose los CFG de los bucles por eficiencia [3]. La Figura 1.9 (Página 11) muestra la CRs calculada con el modelo de coste del número de instrucciones. Esa CR se muestra tras la evaluación parcial (Sección 3.3), por eso la CR de Vector.elimina ya ha sido desplegada y no aparece. Tenemos tres CR recursivas que corresponden a cada bucle de la Figura 1.10: C para el bucle de Lista.delete, D Y E para los bucles while y for de Vector.elimina. Cada variable del programa se abstrae a un tamaño: l es la longitud de lis, $a(b)$ es la longitud del array va.datos (vb.datos), $la(lb)$ es el valor del campo va.size (vb.size) y los parámetros i, j son justamente las variables locales i, j del método Vector.elimina.

- La Ecuación (1) dice que el coste de delete es "1" más el coste del bucle while. Su restricción $a \geq la$ es la interpretación en tamaños del

```
// Lista de enteros implementada con celdas enlazadas.
public class Lista {
    int    valor;        // Valor en esta celda
    Lista next;   // Resto de lista

    static void delete(Lista lis, int piv, Vector va, Vector vb){
        while (lis!=null){ // ecuaciones (2),(3),(4)
            if (lis.valor < piv){
                va.elimina(lis.valor);
            } else {
                vb.elimina(lis.valor);
            }
            lis=lis.next;
        }
    }
}
```

$$
\begin{array}{lllll}
(1) & \langle Del(l,a,la,b,lb) & = & 1+C(l,a,la,b,lb) & \\
 & & , & \{l \geq 0, a \geq la, la \geq 0, b \geq lb, lb \geq 0\}\rangle & \\
(2) & \langle C(l,a,la,b,lb) & = & 2, \{l = 0, a \geq la, a \geq 0, b \geq lb, b \geq 0\}\rangle & \\
(3) & \langle C(l,a,la,b,lb) & = & 25 + D(a,la,0) + E(la,j) + C(l',a,la-1,b,lb) & \\
 & & , & \{l > 0, a \geq la, a \geq 0, b \geq lb, b \geq 0, j \geq 0, l > l'\}\rangle & \\
(4) & \langle C(l,a,la,b,lb) & = & 24 + D(b,lb,0) + E(lb,j) + C(l',a,la,b,lb-1) & \\
 & & , & \{l > 0, a \geq la, a \geq 0, b \geq lb, b \geq 0, j \geq 0, l > l'\}\rangle & \\
(5) & \langle D(a,la,i) & = & 3, \{i \geq la, a \geq la, i \geq 0\}\rangle & \\
(6) & \langle D(a,la,i) & = & 8, \{i < la, a \geq la, i \geq 0\}\rangle & \\
(7) & \langle D(a,la,i) & = & 10 + D(a,la,i+1), \{i < la, a \geq la, i \geq 0\}\rangle & \\
(8) & \langle E(la,j) & = & 5, \{j \geq la-1, j \geq 0\}\rangle & \\
(9) & \langle E(la,j) & = & 15 + E(la,j+1), \{j < la-1, j \geq 0\}\rangle &
\end{array}
$$

Figura 1.9: Definición de Lista y CR de instrucciones de Lista.delete

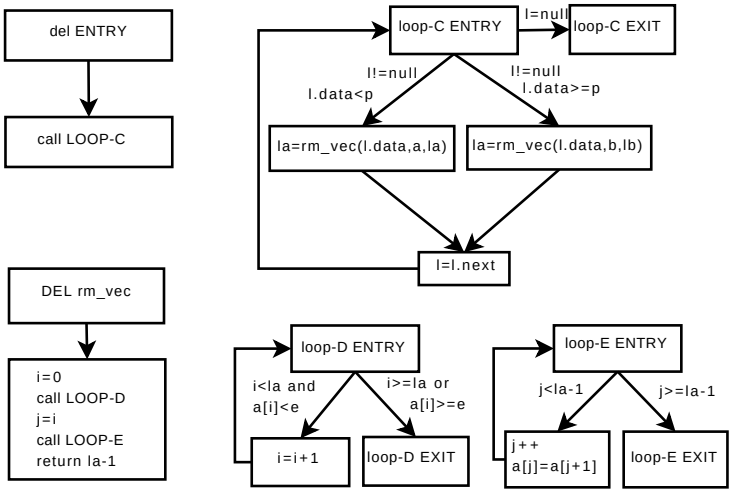

Figura 1.10: Diagrama de Control de Flujo para los bucles del método delete

invariante de Vector: hay entre 0 y length posiciones ocupadas.

- La Ecuación (2) de la relación C corresponde a la salida del bucle while, cuando la lista está vacía (en tamaños $l = 0$).
- Las Ecuaciones (3,4) describen las ramas del if-then-else y solo se diferencian en las variables Vector involucrada (va o vb). Las llamadas a D y E modelan la invocación a Vector.elimina(), tras la que la se decrementa (va se reduce). En cada iteración l disminuye, pero la medida de tamaño usada (longitud de camino) no permite determinar cuánto. Las condiciones l.data $<$ p y l.data $\geq$ p se pierden en el análisis de tamaño. Se usa una variable j para llamar a E porque del valor j de j solo se sabe que $j \geq 0$ tras el bucle while.
- Las Ecuaciones (5,6) son los casos base de D correspondientes a la salida del bucle while, bien porque i $\geq$ la o porque datos[i] $\geq$ e. La condición datos[i] $<$ val se pierde en el análisis de tamaño porque solo sabemos cuánto mide Vector.datos pero no qué vale cada posición datos[i]. La Ecuación (7) es el caso recursivo de D y modela el coste del cuerpo del bucle while. Sus restricciones incluyen la guarda i $<$ size y el efecto de la instrucción i++.
- La Ecuación (8) es el caso base de E que corresponde a la salida del bucle for. La Ecuación (9) el caso recursivo de E que describe el coste de reiterar ese bucle, incrementando j. □

Ejemplo 2 (Evaluación del *CRS* de Lista.delete)**.** La Figura 1.11 (Página 13) muestra dos evaluaciones de $Del(3, 10, 2, 20, 2)$ en el CR de la Figura 1.9, Cada evaluación se representa con una caja con dos partes, a la izquierda el número de la ecuación instanciada y a la derecha el coste local. Las flechas enlazan las subevaluaciones que haya. La evaluación de $Del(v)$ procede así: si $v \leq 0$ entonces aplicamos la ecuación (1) y acumulamos 3 unidades de coste, y si $v > 0$ entonces aplicamos la ecuación (2) lo que acumula 9 unidades más el coste de la llamada a $C(v')$ siendo v' otro entero menor que v. A la izquierda de la figura está la evaluación que da el coste máximo, que escoge siempre la Ecuación (3) de C, y a la derecha la mínima, que usa la Ecuación (4). En las llamadas recursivas asignamos $l' = l - 1$ en la izquierda y $l' = l - 3$ en la de la derecha. Ambas posibilidades son válidas pues cumplen $l' < l$. □

Ejemplo 3 (Imprecisión del análisis de coste)**.** Si se comparan las evaluaciones del Ejemplo 2 con el código del programa (Figura 1.8 en Página 10 y Figura 1.9 en Página 11), solo la evaluación de la izquierda refleja el coste de una ejecución real del programa, pero no así la que está a la derecha. □

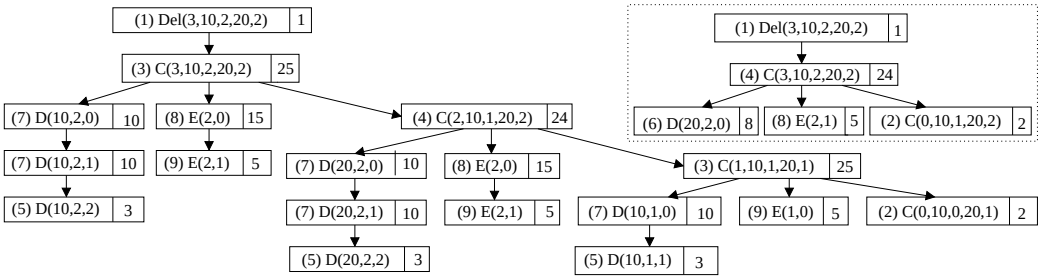

Figura 1.11: Dos evaluaciones para $Del(3, 10, 2, 20, 2)$

Ejemplo 4 (Análisis de ListaBool.m)**.** La Figura1.12 muestra el código del método m, que llama a un método g, y el *CRS* de su número de instrucciones. Las CR C_m y C_g modelan los costes de los métodos m y g respectivamente. La Ecuación (1) corresponde al caso base. Las Ecuaciones $(2, 3)$ modelan las ramas recursivas then y else, y tienen la misma restricción porque this.data no es visible cuando this se abstrae por su longitud. □

```
public class ListaBool {
    private boolean data;     private ListaBool next;
    public void m(i,n){
        if(i<n) {//recursive condition
            if(data){
                g(i,n); next.m(i+1,n);
            } else{
                g(0,i); next.m(i,n-1);
            }
        }
    }
```

$$
\begin{array}{lllll}
(1) & \langle C_m(i,n) & = & 3 & ,\varphi_1 = \{i \geq n\} \ \rangle \\
(2) & \langle C_m(i,n) & = & 15 + C_g(i,n) + C_m(i+1,n) & ,\varphi_2 = \{i < n\} \ \rangle \\
(3) & \langle C_m(i,n) & = & 17 + C_g(0,i) + C_m(i,n-1) & ,\varphi_3 = \{i < n\} \ \rangle
\end{array}
$$

Figura 1.12: Código de la clase ListaBool y CR.

1.3.2. ¿Por qué usar Relaciones de Coste?

Las CR se presentan en el Capítulo 2. Por ahora basta decir que cada una se define con varias ecuaciones de coste(CE). Cada ecuación contiene una expresión (local), llamadas a otras relaciones y una restricción de tamaño que limita el valor de sus variables. Sus características más interesantes son:

No dependen del modelo de coste porque no representan ningún recurso concreto, y pueden ser el lenguaje destino de todos los modelos.

Ejemplo 5 (Independencia de las *CRSs* del modelo de coste)**.** La Figura 1.13 muestra los *CRS* de Vector.suma (Figura 1.8) con los modelos de coste de instrucciones (arriba) y memoria (abajo). Esos *CRS* comparten la misma estructura recursiva, que refleja los bucles del programa, y solo se diferencian en los costes locales que sí vienen determinados por el modelo. □

$$
\begin{array}{lrll}
\langle & suma(0,B) = & 8 & \rangle \\
\langle & suma(A,B) = & 10 + c(A,0,B) \quad , \quad A \geq 1 & \rangle \\
\langle & suma(A,B) = & 26 + c(A,0,B) \quad , \quad A \geq 1 & \rangle \\
\langle & c(A,C,E) = & 0 \quad , \quad C \geq E & \rangle \\
\langle & c(0,C,E) = & 3 \quad , \quad E \geq C+1 & \rangle \\
\langle & c(A,C,E) = & 5 \quad , \quad E \geq C+1, A \geq 1 & \rangle \\
\langle & c(A,C,E) = & 13 + c(A,C+1,E) \quad , \quad E \geq C+1, C \geq 0, A \geq 1 & \rangle
\end{array}
$$

$$
\begin{array}{lrll}
\langle & suma(0,B) = & 0 & \rangle \\
\langle & suma(A,B) = & 0 + c(A,0,B) \quad , \quad A \geq 1 & \rangle \\
\langle & suma(A,B) = & 4 + c(A,0,B) \quad , \quad A \geq 1 & \rangle \\
\langle & c(A,C,E) = & 0 \quad , \quad C \geq E & \rangle \\
\langle & c(0,C,E) = & 0 \quad , \quad E \geq C+1 & \rangle \\
\langle & c(A,C,E) = & 0 \quad , \quad E \geq C+1, A \geq 1 & \rangle \\
\langle & c(A,C,E) = & 0 + c(A,C+1,E) \quad , \quad E \geq C+1, C \geq 0, A \geq 1 & \rangle
\end{array}
$$

Figura 1.13: *CRSs* de Vector.suma() con dos modelos de coste.

Pueden representar muchas clases de complejidad, incluso la del coste ilimitado, porque su estructura recursiva es suficientemente flexible.

Ejemplo 6 (CR de distintos órdenes de complejidad)**.** En la Figura 1.14 se definen varias relaciones de coste: *triang* es una CR de orden cuadrático polinómico, *exp* es una CR de orden exponencial, *log* es una CR de orden logarítmico e $infty$ es una CR de recursión infinita. □

$$
\begin{array}{llcll}
\langle & triang(1) & = & 1 & \rangle \\
\langle & triang(n) & = & nat(n) \;\; +triang(n-1) & , \;\; n \geq 2 \;\; \rangle \\
\langle & exp(1) & = & 1 & \rangle \\
\langle & exp(x) & = & 1 \;\; +exp(x-1)+exp(x-1) & , \;\; x \geq 1 \;\; \rangle \\
\langle & log(1) & = & 1 & \rangle \\
\langle & log(x) & = & 1 \;\; +binary(\tfrac{x}{2}) & , \;\; x \geq 2 \;\; \rangle \\
\langle & infty(x) & = & 1 \;\; +infty(x) & \rangle
\end{array}
$$

Figura 1.14: Ejemplos de CR de distintas clases de complejidad.

Como no dependen del lenguaje fuente, el mismo resolutor de segunda fase sirve para un analizador de primera fase de JAVA, PROLOG, HASKELL, C, etc. Además hacen que los resultados de la primera fase sean *casi los mismos* para varias implementaciones de un algoritmo en cada lenguaje (Figura 1.15).

```
public Lista merge(Lista xs, Lista ys){
    if(xs == null){
        return ys;
    } else if(ys == null){
        return xs;
    } else if(xs. valor <= ys.valor){
        return new Lista(xs.valor, merge(xs.next,ys))
    } else {
        return new Lista(ys.valor, merge(xs,ys.next));
    }
}
```

```
merge(Xs,[],Xs) :- !.
merge([],Ys,Ys) :- !.
merge([X|Xs], [Y,Ys], [X,Rs]) :- X<=Y, !, merge(Xs,[Y|Ys],Rs).
merge([X|Xs], [Y,Ys], [Y,Rs]) :- X >Y, !, merge([X|Xs],Ys,Rs).
```

```
merge xs []      = xs
merge [] xs      = xs
merge (x:xs) (y:ys)     | x<= y = x: (merge xs (y:ys))
                        | x > y = y: (merge (x:xs) ys)
```

$$
\begin{array}{llcll}
(1) & \langle merge(A,0) & = & 4 \;\; , \;\; \{A \geq 0\} & \rangle \\
(2) & \langle merge(0,B) & = & 4 \;\; , \;\; \{B \geq 1\} & \rangle \\
(3) & \langle merge(A,B) & = & 26 + merge(A,B') \;\; , \;\; \{A \geq 1, B \geq 1, B' = B-1\} & \rangle \\
(4) & \langle merge(A,B) & = & 26 + merge(A',B) \;\; , \;\; \{A \geq 1, B \geq 1, A' = A-1\} & \rangle
\end{array}
$$

Figura 1.15: Implementación JAVA, PROLOG y HASKELL de Merge, y *CRS* común a las tres implementaciones.

1.4. Resolución de Relaciones de Coste

Una CR es más simple que el programa cuyo coste modela. Pero son (a) recursivas, por cuanto su evaluación es iterativa, e indeterministas pues cada llamada $C(\overline{x})$ representa varios valores. Esto las hace demasiado complejas para las aplicaciones prácticas mencionadas en la Sección 1.2. En éstas se necesitan resultados en **forma cerrada** *(closed form)*, esto es no recursivos y deterministas, como por ejemplo cotas superiores o inferiores.

La segunda fase de la arquitectura de Wegbreit consiste en calcular resultados en forma cerrada para una CR. Mientras que la primera fase está estudiada en detalle, la segunda ha recibido bastante menos atención. La falta de herramientas para resolver CR posiblemente sea el principal obstáculo para el empleo del análisis automático de coste.

1.4.1. Diferencias entre las CR y las *RR*?

Los **sistemas de álgebra por computador** *(Computer Algebra Systems, CAS)*, como Mathematica®, MAXIMA, MAPLE y otros, implementan algoritmos para resolver las tradicionales **Relaciones de Recurrencia** *(Recurrence Relations RR)* que son usadas en otras ramas de ciencia e ingeniería. Esta Subsección muestra qué diferencias hay entre las *RR* y las CR que generan los analizadores que impiden usar los CAS para resolver las CR.

En esencia, una *RR* define una función $f : \mathbb{N}^n \to \mathbb{R}^+$ mientras que una CR define una relación $C : \mathbb{N}^n \to \mathcal{P}(\mathbb{R}^+)$. Para un vector $\bar{v}$, $f(\bar{v})$ es un valor pero $C(\bar{v})$ es un conjunto de valores.

Hay que admitir que con una *RR* se pueden representar recurrencias más complicadas que con una CR. Esta capacidad adicional no tiene mucho valor en el análisis de coste porque los programas iterativos raramente contienen bucles que se modelen con esas iteraciones.

Ejemplo 7 (*RR* que no equivale a ninguna CR)**.** La función f definida con la *RR* $f(x, y) = x^2 + f(x^2 - y^2, x - y)$ no es representable con una CR. □

Las ecuaciones de una CR no son mutuamente excluyentes: eso permite que $C(\overline{v})$ pueda contener varios resultados, lo que permite modelar en una CR el indeterminismo inherente al programa fuente o al análisis de coste. Lo primero se refiere a instrucciones que no se pueden evaluar estáticamente, como la entrada y salida, el azar, polimorfismo en un lenguaje orientado a objetos, etc. El segundo se refiere a la información que se pierde al realizar la abstracción de tamaño, como los valores de los campos de un registro o las posiciones de un array.

Ejemplo 8 (Evaluación indeterminista de una CR)**.** En el *CRS* de la Figura 1.9 las Ecuaciones (3) y (4) comparten la misma restricción. Lo mismo sucede entre las Ecuaciones (6) y (7). □

Esta cuestión ya la comentó Wegbreit en [57], y proponía afrontarlo añadiendo un operador `when` al lenguaje *RR*, pero esto solo servía en casos muy simples. En muchos *frameworks*, el primer paso es convertir las CRs en *funciones de coste*. Para ello intentan suprimir el indeterminismo modelando las ecuaciones para dirigirlas a un caso de interés (pésimo u óptimo).

Las restricciones inexactas de las ecuaciones de coste no determinan unívocamente las variables en función de los parámetros de entrada: pueden restringirlas con una inecuación $l \geq 0$ o pueden tomar cualquier valor.

Ejemplo 9 (Restricción inexacta)**.** En la Figura 1.9, en las ecuaciones (3,4) l' puede tomar cualquier valor entero que cumpla $l > l'$. □

Esto se necesita para modelar bucles sobre estructuras no lineales de datos: en un árbol solo se puede inferir que cada subárbol tiene una altura estrictamente menor. Aunque no sea evidente en ejemplos pequeños, es inevitable encontrarlo al analizar programas que manejan árboles, o cuando el análisis de los valores numéricos pierde precisión.

Argumentos múltiples. En una llamada recursiva, cada parámetro de la relación puede crecer o disminuir concurrentemente con los otros. Por eso el número de iteraciones puede depender de varios argumentos.

Ejemplo 10 (Argumentos múltiples de bucles)**.** En la Figura 1.9, en la Ecuación (2) l se reduce mientras que en (7) i aumenta, y por eso el número de iteraciones de E es $la - j - 1$. □

La mayoría de los CAS (excepto Mathematica®) solo manejan recurrencias de una variable. A veces es posible descomponer automáticamente una CR de varios parámetros en varias relaciones de un argumento. Pero este camino lleva a resultados correctos solo si todas las ecuaciones de la CR tienen un coste local constante, lo que solo pasa en un CR de juguete.

Tratamiento

Dada una CR indeterminista, a veces resulta útil definir una *cost bound function (CBF)*. Una forma relativamente trivial para obtenerla sería introducir un operador de maximización, pero los CAS existentes no soportan

dicho operador. Y resulta complicado añadirlo porque como hay varias ecuaciones no exclusivas, generar el conjunto de soluciones para calcular su máximo es un problema combinatorio intratable. Esto también afecta al uso de tales CBF en enfoques dinámicos como el de Gómez et al. [28].

Otro enfoque para obtener una CBF es quitar el indeterminismo de la CR, para lo que suprime algunas ecuaciones y escoge la solución pésima de las restricciones inexactas. Esto no siempre es correcto porque en algunos *CRS* el caso pésimo corresponde a una evaluación que entrelace ecuaciones distintas, y si se suprime una de ellas entonces el nuevo CR admite cotas superiores que no valen para la original.

Ejemplo 11 (Evaluación Pésima de CR)**.** En la Figura 1.9, el caso pésimo es la evaluación que usa alternativamente las Ecuaciones (3,4). □

1.4.2. Trabajos Relacionados

Como se dijo en la Sección 1.3, este proyecto sigue el enfoque de análisis automático de coste del artículo germinal de Wegbreit [57]. Existen varios analizadores de coste que intentan construir funciones de cota superior, para lenguajes funcionales [57, 38, 50, 56, 53, 16, 42], lógicos [27, 45] e imperativos [7]. Como no hay una terminología unificada en el área, esas funciones son conocidas como **funciones de complejidad de caso pésimo** *(worst-case complexity functions WCCF)* por Aho, Hopcroft y Ullman [1], como **funciones de restricción de tiempo** *(time-bound functions TBF)* por Rosendahl [50], y como **funciones recursivas de complejidad en el tiempo** *(recursive time-complexity functions RTCF)* por Le Metayer [38]. Como ya hemos dicho en la Sección 1.4.1), las CR se diferencian de todas ellas en que

1. las ecuaciones llevan restricciones de tamaño inexactas asociadas y
2. especifican relaciones indeterministas.

Hay dos maneras de ver y resolver las CR:

La perspectiva algebraica considera las CRs como relaciones de recurrencia *(recurrence relations)*. Fue la propuesta inicial, y tiene gran apoyo porque permite reutilizar el conocimiento existente en resolución de relaciones de recurrencia, de dos modos alternativos:

a implementar en el analizador un resolutor de recurrencias restringidas *(restricted recurrence)*, como hacen Debray [27] y Wegbreit [57].

b Acoplar un sistema CAS a la salida del analizador, tal y como propusieron Benzinger [16], Luca et al. [42], Sands [53] y Wadler [56].

La perspectiva transformacional considera una CRs como un programa funcional, y las resuelve aplicando técnicas genéricas de transformación de programas sobre el **programa de tiempo acotado** *(time bound program TBP)* [50] hasta obtener un programa no recursivo. Resulta trivial obtener un TBP a partir de un CRs introduciendo un operador de maximización (o ejecución disjunta). Esta perspectiva fue propuesta primero en ACE [38], que contenía varias reglas de transformación para obtener representaciones no recursivas. También fue defendida por Rosendahl [50], quien más tarde proporcionó varias técnicas [51] basadas en la súper compilación (Turchin [55]).

Discusión

La necesidad de mejorar los algoritmos para calcular resultados en forma cerrada ya fue señalada por Hickey y Cohen [32]. Un avance significativo en esta dirección es el de Roberto Bagnara en PURRS [13], que fue el primero en calcular algorítmicamente cotas superiores en forma cerrada no asintótica para una amplia clase de *CRS*, pero desafortunadamente necesita que éstos sean deterministas.

Otro trabajo relevante es el de Marion et al. [43, 17], quienes propusieron un análisis para lenguajes funcionales que acotaba el tamaño de la pila de registros de activación. Este enfoque usaba las cuasi-interpretaciones y estaba limitado a la complejidad polinómica.

Una alternativa a la búsqueda de cotas superiores es evaluar directamente las TBF como proponían Gómez et al. en [28], si bien esto tiene un área de aplicación menor y su uso eficiente está restringido a sistemas de ecuaciones deterministas.

Trabajos que no siguen el modelo de Wegbreit

Otro enfoque diferente al análisis de coste se basa en los sistemas de tipos anotados con información de recursos, sin usar las CR como paso intermedio. Así, en este enfoque no se necesita el cálculo de resultados en forma cerrada para las CR, pero está restringido a cotas lineales (Hoffman [33]), con alguna notable excepción como en el trabajo de Crary y Weirich [26].

Un enfoque basado en el análisis de programas para inferir el acotamiento polinómico de los valores calculados (como función de la entrada) ha sido propuesto por Ben-Amram et al. [15]. Infiere la complejidad de un programa obteniendo primero un *step-counting program*. Este trabajo se basa en trabajos previos en la línea de Niggl et al. [47] y Kristiansen et al.[37] y la mayor novedad es que proporciona completitud para un lenguaje simple (Turing incompleto).

1.4.3. Nuestro enfoque

El problema de las metodologías expuestas era su limitada escalabilidad: podían aplicarse con éxito en el análisis de programas de juguete, pero no en el de programas reales. En unas por su incapacidad de tratar con éxito el indeterminismo, en otras porque solo analizaban programas de cierto lenguaje o paradigmas, y en algunas porque solo podían tratar programas cuyos costes tuvieran un orden de complejidad lineal o polinómico.

La experiencia de Albert, Arenas, Genaim, Puebla y Zanardini en el uso del enfoque algebraico para resolver las CR generadas por COSTA [7] evidenció que los CAS no resuelven o no pueden resolver con éxito una CR, y que convertir automáticamente los *CRS* al formato de los CAS o bien no es posible o bien lleva a soluciones inútiles para uso práctico (Sección 1.2).

En cambio el enfoque que aquí si sigue logra calcular resultados en forma cerrada correctos y relativamente simples. Su aplicabilidad abarca CR deterministas o indeterministas, programas funcionales, lógicos u orientados a objetos, y de las clases de complejidad polinómica, exponencial o logarítmica. Una característica distintiva de este enfoque es que no traduce las CR a otro formato. En cambio, las procesamos y simplificamos hasta derivar estáticamente en una solución en forma cerrada. Para ello aplicamos análisis y transformaciones basadas en la semántica como el cálculo de funciones de rango de Podelski [48], los invariantes (Cousot y Halbwachs [24]) y la evaluación parcial (hay una monografía de Jones et al. [35]). Estos análisis han sido estudiados en el campo de la complejidad algorítmica (Wilf [58]).

En síntesis, este enfoque es más potente pero las técnicas que emplea son inherentemente incompletas.

1.5. La perspectiva asintótica

El mismo Wegbreit [57] señaló que los analizadores generan funciones sintácticamente grandes y complejas. Para algunas aplicaciones es deseable abstraer la información más relevante y suprimir los detalles innecesarios. Un método es el *análisis asintótico*, que se basa en estas suposiciones:

- Solo importa el comportamiento de las funciones para valores indefinidamente grandes (límite infinito o asintótico)
- Al compararlas, podemos ignorar las proporciones lineales.

La notación $\mathcal{O}$ denota *cotas superiores asintóticas*. Dadas $f, g : \mathbb{N} \to \mathbb{R}$, $f \in \mathcal{O}(g)$ expresa que para valores grandes de n , $f(n)$ está proporcionalmente acotada por $g(n)$. Otras notaciones asintóticas son la notación $\Omega(f)$, para cotas inferiores y la notación $\Theta(f)$ para proporción lineal.

1.5.1. Análisis asintótico de coste

El **análisis asintótico de coste** *(asymptotic cost analysis)* busca encontrar funciones sencillas que describan el comportamiento del programa para datos grandes. Las funciones f'_P, max_P, min_P, definidas en las ecuaciones (1.2), en la página 4 permiten distinguir dos tipos de estudios asintóticos:

El estudio de caso pésimo *(asymptotic worst-case)* busca una función ub tal que $max_P \in \mathcal{O}(ub)$, llamada **cota superior asintótica** *(asymptotic upper bound AUB)*. Si esa función ub también satisface que $max_P \in \Theta(ub)$, entonces ub es una **cota superior asintótica exacta** *(tight asymptotic u.b.)*.

Encontrar una cota inferior asintótica *(asymptotic lower bound)*, esto es una función lb tal que $min_P \in \Omega(lb)$, es el objetivo del estudio de **caso óptimo** *(asymptotic best-case)*. lb será una **cota inferior asintótica exacta** *(tight asymptotic lower bound)* si $min_P \in \Theta(lb)$.

El orden asintótico exacto del programa P lo da una función f que sea a la vez cota asintótica superior e inferior del coste del programa. Es decir $max_P \in \mathcal{O}(f)$ y $min_P \in \Omega(g)$ y $f \in \Theta(g)$.

La Complejidad Computacional asintótica es el estudio del comportamiento a gran escala de un algoritmo, o de los algoritmos que resuelven un problema, empleando análisis asintótico. Desde el artículo fundamental

de Hartmanis y Stearns[1] en 1965, así como del libro de 1972 sobre NP-completitud de Garey y Johnson, también se usa el término **complejidad computacional** para referirse a la complejidad asintótica.

1.5.2. ¿Asintóticos o no asintóticos?

Los dos tipos de análisis son complementarios y ninguno es adecuado para todas las aplicaciones mencionadas en la Sección 1.2. El primero da información esencial y sencilla pero más imprecisa que el segundo. Siempre se puede pasar de no asintótico a asintótico, de forma rápida y sencilla, pero este proceso destruye información y es irreversible. Por ello siempre es más útil y correcto calcular cotas no asintóticas. Y hay aplicaciones, como predecir si una aplicación satura los recursos del sistema (Gómez, Albert y Genaim [9]) en que solo valen las cotas no asintóticas. No obstante, hay dos grandes motivos para usar resultados asintóticos.

Por legibilidad: En la Subsección 1.2.1 se mencionó cómo puede un programador usar el análisis de coste. Para lograrlo, se debe facilitar la interacción y presentación de los datos. Pero las expresiones que se obtienen en el análisis no asintótico, como $5x^2 + 3x * log_2 x + 2x + 1$ son poco legibles y excesivamente detalladas, mientras que x^2 es más conciso.

Por eficiencia de cálculo: Los analizadores de coste también son programas y por tanto nos interesa reducir *sus* costes. El análisis asintótico es más eficiente que el no asintótico porque éste procesa expresiones más compactas, ignora las constantes aditivas y multiplicativas y puede resolver con más facilidad los operadores de máximo o mínimo.

1.6. Nuestra contribución

Se adapta la noción clásica de *complejidad asintótica* para cubrir el análisis de programas reales. En ellos el comportamiento asintótico viene determinado por el número de iteraciones de los bucles, que dependen de combinaciones lineales de varios argumentos del programa.

Hemos implementado una simplificación de expresiones de coste a forma asintótica equivalente. Esto satisface la motivación (1) y nos permite usar el conocimiento y arte existente en resolución de CRs.

Esa simplificación la hemos integrado en el cálculo de resultados en forma cerrada, para que así se calculen directamente cotas asintóticas sin obtener primero las no asintóticas. Esto satisface nuestra segunda motivación.

Capítulo 2

Sistemas de Relaciones de Coste

2.1. Notaciones

Empecemos por introducir alguna notación:

- Conjuntos de números naturales $\mathbb{N}$, naturales mayores que cero $\mathbb{N}^+$ enteros $\mathbb{Z}$, reales $\mathbb{R}$, y reales no negativos $\mathbb{R}^+$.
- $\bar{t}$ denota una secuencia de $n > 0$ elementos $t_1, \ldots, t_n$.
- x, y, o z, denotan variables de dominio $\mathbb{Z}$.
- Para unas variables $\bar{x} = (x_1, ..., x_n)$, una **sustitución** σ es una aplicación $\sigma : \bar{x} \mapsto \mathbb{Z}^n$, a veces representado como $\bar{x} = \bar{v}$ siendo $\bar{v} = (v_1, ..., v_n)$ un vector de valores enteros.
- Una **expresión lineal** *(linear expression)* tiene la forma $v_0 + v_1x_1 + \ldots + v_nx_n$, donde $v_i \in \mathbb{Z}$, $0 \leq i \leq n$.
- Así mismo, una **restricción lineal** *(linear constraint)* (en $\mathbb{Z}$) es una fórmula $l \leq 0$, donde l es una expresión lineal. Abreviaremos $l = 0$ por $l \leq 0 \wedge l \leq 0$, y $l < 0$ por $l + 1 \leq 0$.
- Una **restricción de tamaño** *(size constraint SC)* es una conjunción de restricciones lineales. Las letras griegas φ, ϕ o ψ simbolizan SC.
- $\varphi_1 \models \varphi_2$ indica que φ_1 implica φ_2. Asimismo, $\sigma \models \varphi$ indica $\bar{x} = \bar{v} \models \varphi$.
- El operador de proyección $\exists\bar{x}.\varphi$ (respectivamente $\bar{\exists}\bar{x}.\varphi$) suprime de φ las variables en $\bar{x}$ (respectivamente $vars(\varphi) \setminus \bar{x}$).

2.2. Expresiones de Coste

Esta sección describe las expresiones de coste, que son los bloques básicos con que se construyen las relaciones de coste. Definimos su sintaxis y sus propiedades semánticas, las cuales proporcionan el fundamento del cálculo de cotas superiores de relaciones de coste.

2.2.1. Sintaxis

Definición 5 (Expresión de Coste)**.** Una **Expresión de Coste** *(Cost Expression CExp)* es una expresión simbólica e que puede generarse con esta gramática:

$$e ::= r \mid \mathsf{nat}(l) \mid \log(\mathsf{nat}(l)) \mid n^{\mathsf{nat}(l)} \mid e + e \mid e * e \mid e^r \mid \text{máx}(S)$$

donde $r \in \mathbb{R}^+$, $n \in \mathbb{N}^+$, l es una expresión lineal, la base del logaritmo log es 2 y S es un conjunto de expresiones de coste. □

Para una CExp e definimos sus conjuntos de subexpresiones lineales $lins(e)$ y variables $vars(e)$. Obsérvese que siempre están dentro de un nat.

Ejemplo 12 (Variables y subexpresiones lineales)**.** En la expresión de coste $e = \mathsf{nat}(x+1) * \mathsf{nat}(y-2) + 3 * \mathsf{nat}(y-3)$ los conjuntos $lins(e)$ y $vars(e)$ son $lins(e) = \{x+1, y-2, y-3\}$ y $vars(e) = \{x, y\}$. □

Definición 6 (Expresión de Coste Básica)**.** Una **Expresión de Coste Básica** *(Base Cost Expression)* es una expresión de coste con la forma $r, \mathsf{nat}(l), \log(\mathsf{nat}(l))$ o $n^{\mathsf{nat}(l)}$. □

Ejemplo 13 (Expresión de Coste Básica)**.** En estos grupos de expresiones:

$$5 \qquad \mathsf{nat}(2x+1) \qquad 3^{\mathsf{nat}(y-1)} \qquad log_3(\mathsf{nat}(z+2)+1) \tag{2.1}$$

$$5+1 \qquad \mathsf{nat}(x) + \mathsf{nat}(y) \qquad 2^{\mathsf{nat}(z)} * \mathsf{nat}(x) \qquad \text{máx}(\{\mathsf{nat}(x), 5\}) \tag{2.2}$$

las expresiones en (2.1) son básicas, mientras que las de (2.2) no lo son. □

Las expresiones de coste no básicas se obtienen aplicando las operaciones $+, *,$ máx sobre las básicas. O dicho de otra manera, **en el árbol sintáctico de una CExp los nodos hoja son expresiones básicas.**

2.2.2. Semántica

El propósito de una expresión de coste es definir una función entre los tamaños de los datos de entrada de un programa y el coste de procesar esos datos. Su semántica consiste en especificar cómo se evalúa.

Definición 7 (Evaluación de una Expresión de Coste)**.** Sea e una expresión de coste con variables libres $vars(e) \subseteq \bar{x} = (x_1, \ldots, x_n)$. Y sea una asignación $\sigma : \bar{x} \mapsto \mathbb{Z}^n$. La **evaluación** de e con σ, escrito $\sigma(e)$ o $[\![e']\!]_\sigma$, es el número que se obtiene siguiendo estos pasos:

1. Aplicamos σ en todas las variables en e.
2. Interpretamos nat como una función $\mathsf{nat}(v)$=$max(\{v, 0\})$.
3. El resto de operaciones (suma, producto, exponente, max) se interpretan de manera convencional, y se operan. □

Esta definición garantiza que al evaluar una expresión de coste siempre se obtiene un valor positivo o nulo.

Ejemplo 14 (Evaluación de una Expresión de Coste)**.** La evaluación de esta expresión de coste,

$$e = 5 + \mathsf{nat}(x) * (\mathsf{nat}(y) + 1) + 2^{\mathsf{nat}(z-1)}$$

con la sustitución σ definida como $\sigma(x) = 0$, $\sigma(y) = 2$ y $\sigma(z) = 1$, procedería así:

$$\begin{aligned} [\![e]\!]_\sigma &= \\ [\![5 + \mathsf{nat}(x) * (\mathsf{nat}(y) + 1) + 2^{\mathsf{nat}(z-1)}]\!]_\sigma &= // \text{ aplicar } \sigma \\ 5 + \mathsf{nat}(0) * (\mathsf{nat}(2) + 1) + 2^{\mathsf{nat}(0)} &= // \text{ aplicar } \mathsf{nat} \\ 5 + 0 * (2 + 1) + 2^0 &= 6 \text{ operar} \end{aligned}$$

□

2.2.3. Equivalencia y Forma Normal

Para tratar las expresiones de coste, hay que empezar por señalar cuándo dos expresiones sintácticamente distintas representan la misma función.

Definición 8 (Equivalencia de Expresiones de Coste)**.** Sean e_1, e_2 dos CExp tales que $vars(e_1) \cup vars(e_2) = \{a_1, \ldots, a_n\}$. Decimos que e_1 y e_2 son equivalentes, escrito $e_1 = e_2$, si para cualquier asignación σ de $\{a_1, \ldots, a_n\}$ en $\mathbb{R}^+$ se cumple que $\sigma(e_1) = \sigma(e_2)$. Se puede comprobar que "= "es una relación de equivalencia: es reflexiva, simétrica y transitiva. □

Propiedad 1 (Propiedades de equivalencia entre CExp)**.** Con respecto a "= ", las operaciones entre expresiones de coste se comportan de la manera usual: El operador de suma es conmutativo y asociativo y tiene un elemento neutro (0) (2.3). El operador de producto es conmutativo y asociativo y tiene elemento neutro (1) (2.4). El operador de máx ($\sqcup$) es conmutativo, asociativo y tiene elemento neutro (0) (2.5). El producto es distributivo respecto de la suma (2.6) y tanto la suma como el producto lo son respecto del máx(2.7). Por último, la suma y el producto son cancelativas (2.8). □

$$a + (b + c) = (a + b) + c \qquad a + b = b + a \qquad a + 0 = a \tag{2.3}$$
$$a * (b * c) = (a * b) * c \qquad a * b = b * a \qquad a * 1 = a \tag{2.4}$$
$$a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c \qquad a \sqcup b = b \sqcup a \qquad a \sqcup 0 = a \tag{2.5}$$

$$a * (b + c) = a * b + a * c \tag{2.6}$$
$$a + (b \sqcup c) = a + b \sqcup a + c \qquad a * (b \sqcup c) = a * b \sqcup a * c \tag{2.7}$$
$$a + b = a + c \rightarrow b = c \qquad a * b = a * c \rightarrow a = 0 \vee b = c \tag{2.8}$$

Definición 9 (Forma Normal de Expresiones de Coste)**.** Una expresión de coste e está en forma normal si tiene esta estructura

$$e = \bigsqcup_{i=1}^{m} \sum_{j=1}^{s_i} \prod_{k=1}^{p_{ij}} b_{ijk}$$

donde cada b_{ijk} es una expresión de coste básica. □

Proposición 1 (Normalización de Expresiones de Coste)**.** Para toda CExp e existe otra expresión en forma normal e' tal que $e = e'$

Demostración. Para una expresión de coste e, su forma normal equivalente se puede construir aplicando las propiedades asociativas y conmutativas de $+, \times$ y máx, así como las propiedades distributivas del producto respecto a la suma y las de éstas respecto del máx. □

Ejemplo 15 (Normalización de Expresión de Coste)**.** Vamos a normalizar la siguiente expresión $\mathsf{nat}(x) * (5 + \text{máx}(\{\mathsf{nat}(y)^2, 2^{\mathsf{nat}(z)}\}))$

$$\begin{aligned} & \mathsf{nat}(X) \underbrace{* \text{máx}(\{5 + \mathsf{nat}(y)^2, 2^{\mathsf{nat}(z)}\})}_{\text{distributividad de } * \text{ sobre } \sqcup} = \\ \text{máx}(\{\mathsf{nat}(x) & \underbrace{*(5 + \mathsf{nat}(y)^2)}_{\text{distributividad de } * \text{ sobre } +}, \mathsf{nat}(x) * 2^{\mathsf{nat}(z)}\}) = \\ & \text{máx}(\{5\mathsf{nat}(x) + \mathsf{nat}(x) * \mathsf{nat}(y)^2, \mathsf{nat}(x) * 2^{\mathsf{nat}(z)}\}) \end{aligned}$$

en cada paso se indica qué propiedad distributiva es la utilizada. □

2.2.4. Orden y Monotonía

Uno de nuestros objetivos en análisis de coste es comparar las funciones de coste de varios programas. Dado que las expresiones de coste representan funciones de coste, debemos definir un orden entre ellas.

Definición 10 (Orden funcional de CExp)**.** Sean e_1, e_2 dos expresiones de coste tales que $vars(e_1) \cup vars(e_2) = \{a_1, \ldots, a_n\}$. Decimos que e_1 es menor o igual que e_2, escrito $e_1 \leq e_2$, si para cualquier asignación σ de $\{a_1, \ldots, a_n\}$ a $\mathbb{R}^+$ se cumple que $\sigma(e_1) \leq \sigma(e_2)$. □

Es decir, $e_1 \leq e_2$ indica que e_1 acota inferiormente a e_2. Podemos comprobar que "$\geq$"es una relación de orden: es reflexiva, transitiva, antisimétrica con respecto a "="y tiene un elemento mínimo (0). Los operadores $+, *,$ máx se comportan de la manera usual respecto a "$\geq$".

Propiedad 2 (Propiedades del orden funcional $\leq$)**.** El operador de suma es extensivo y monótono(2.9). El operador de producto es monótono y, cuando todos los factores son mayores que 1, extensivo(2.10). El operador de máx es monótono y extensivo(2.11). Además, máx es un operador de de **mínima cota superior** *(least upper bound)* (2.12). □

$$a \leq b \rightarrow a + c \leq b + c \qquad a + b \geq a \tag{2.9}$$

$$a \leq b \rightarrow a * c \leq b * c \qquad b \geq 1 \rightarrow a * b \geq a \tag{2.10}$$

$$a \leq b \rightarrow a \sqcup c \leq b \sqcup c \qquad a \sqcup b \geq a \tag{2.11}$$

$$a \sqcup b = c \leftrightarrow a = c \wedge a \geq b \vee b = c \wedge b \geq a \tag{2.12}$$

Otra propiedad que también es importante señalar es la monotonía de las expresiones de coste respecto a sus componentes lineales.

Proposición 2 (Monotonía de Expresiones de Coste)**.** Sea e una CExp con una subexpresión lineal l, una expresión lineal l' y φ una fórmula tal que $\varphi \models l' \geq l$. Sea e' el resultado de reemplazar dentro de e una aparición de $\mathsf{nat}(l)$ por $\mathsf{nat}(l')$. Entonces se cumple que $e' \geq e$. □

Demostración. Se infiere de la monotonía de los operadores nat, potencia y logaritmo, así como de las operaciones $+, \times$,máx. □

Ejemplo 16 (Monotonía de Expresiones de Coste)**.** Tomemos la CExp $e = \mathsf{nat}(x)+5$ y sea la restricción de coste $\varphi = \{x \leq y\}$. Resulta entonces evidente que la expresión $e' = \mathsf{nat}(y) + 5$ verifica que $\varphi \models e' \geq e$. □

2.3. Sintaxis de los *CRS*

Esta sección se define la sintaxis de las CR y los *CRS*, identifica sus componentes principales y define algunas propiedades importantes.

2.3.1. Ecuaciones de Coste

Definición 11 (Ecuación de Coste)**.** Una **ecuación de coste** *(cost equation, CE)* es una fórmula

$$\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi \rangle$$

donde e es una CExp, llamada **coste local**, C es la relación de coste codefinida por la ecuación, $\bar{x}$ son los **parámetros** de la ecuación, cada D_i es una relación de coste, las expresiones $D_i(\bar{y}_i)$ son **llamadas** y φ una restricción de tamaño entre las variables $\bar{x} \cup vars(e) \bigcup_{i=1}^{k} \bar{y}_i$, llamada **guarda**. □

Intuitivamente, la ecuación $\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi \rangle$ dice que el coste de $C(\bar{x})$ es el valor de e (coste local) más la suma de los valores de las llamadas $D_i(\bar{y}_i)$ para una asignación de $vars(e), \bar{x}, \bar{y}_i$ que satisface φ. Se puede ver que, a diferencia de las ecuaciones en una *RR*, la guarda de una CE es un sistema de inecuaciones $l \leq 0$ o $l < 0$.

Una ecuación **llama** a una CR D si contiene alguna llamada a D. Una llamada es **recursiva** *(recursive)* si $D = C$ y externa en otro caso.

Ejemplo 17 (Ecuación de Coste y Elementos)**.** Esta CR

$$\langle C(x, y) = \mathsf{nat}(x-y)+4+D(x+2, z, 1)+C(x, y-1), \{x \geq 0, y \geq 0, 0 \leq z \leq y\} \rangle$$

solo contiene una ecuación de la CR C cuyos parámetros son x, y. Esta CE contiene un coste local que es $\mathsf{nat}(x - y) + 4$, una llamada a la CR D y una llamada recursiva. La restricción inexacta indica que esta CE solo es aplicable si los parámetros x, y tienen un valor positivo o 0 y que la otra variable z solo toma valores entre 0 e y (ambos incluidos). □

Como se vio en el Capítulo 1, nuestro objetivo es transformar las CR hasta conseguir resultados en forma cerrada. Por eso vamos a especificar ahora qué es esa forma cerrada.

Definición 12 (Formas Directa, Funcional y Cerrada de una Ecuación de Coste)**.** Una CE está en **forma directa** *(direct form)* si no contiene llamadas, está en **forma funcional** *(functional form)* si los parámetros determinan unívocamente a todas sus variables y está en **forma cerrada** *(closed form)* si está en forma directa y funcional. □

Ejemplo 18 (Ecuaciones en forma funcional y directa)**.** En estas CE

$$\begin{array}{lll} (1)\ \langle C(x,y) = & 5 + D(z) & ,\{z \leq y\}\rangle \\ (2)\ \langle C(x,y) = & 5 + \mathsf{nat}(x-z) & ,\{z \leq y\}\rangle \\ (3)\ \langle C(x,y) = & 3 + D(x-y) & ,\{x \geq 0\}\rangle \\ (4)\ \langle C(x,y) = & 3 + \mathsf{nat}(x-y) & ,\{x \geq 0\}\rangle \end{array}$$

1. (1) no está en forma ni funcional ni directa,
2. (2) está en forma directa pero no funcional,
3. (3) está en forma funcional no directa y
4. (4) está en forma funcional y directa, y por tanto en forma cerrada. □

Ahora vemos como caracterizar la recursividad de las relaciones de coste a nivel de ecuación.

Definición 13 (Recursión de las Ecuaciones de Coste)**.** Una CE $\mathcal{E}$ es una **ecuación básica** *(base equation)*, o caso base *(base case)* si no contiene llamadas recursivas y es una **ecuación recursiva si** *(recursive equation)*. El **factor recursivo** *(recursive factor)* de una CE es el número de llamadas recursivas que contiene. Una ecuación recursiva es **recursiva simple** si solo tiene una llamada recursiva, y **recursiva múltiple** si tiene más de una. □

Ejemplo 19 (Ecuaciones básicas y recursivas)**.** En estas CE

$$\begin{array}{llllll} (1) & \langle C(x) = & 1 & + & D(y) & , \quad x \geq y \geq 0 \quad \rangle \\ (2) & \langle C(x) = & 1 & + & C(x-1) & , \quad x \geq 2 \quad\ \ \rangle \\ (3) & \langle C(x) = & 1 & + & C(x-1) + C(x-2) & , \quad x \geq 3 \quad\ \ \rangle \end{array}$$

(1) contiene una llamada externa pero ninguna recursiva, por lo que se trata de un caso base, (2) es recursiva simple con factor recursivo $b = 1$ y (3) es una ecuación recursiva múltiple de factor $b = 2$. □

El factor recursivo es una medida fundamental para describir la evaluación de una relación de coste. Cuando existe recursión múltiple, la evaluación de la CR implica un número exponencial de evaluaciones derivadas. Por tanto, la recursión múltiple es el origen de los órdenes exponenciales de complejidad.

2.3.2. Relaciones de Coste

Definición 14 (Relación de Coste)**.** Una **relación de coste** *(cost relation, CR)* de aridad n es una relación $C \subseteq \mathbb{Z}^n \times \mathbb{R}^+$ especificada por varias CE. □

La forma en que las ecuaciones de coste especifican los valores de una CR se detalla en la Sección 2.4.

Ejemplo 20 (Relación de Coste)**.** Estas dos ecuaciones de coste

$$\begin{array}{llll} \langle C(x,y) = 2 * \mathsf{nat}(y) + 3 & , & \{x = 0, y \geq 0\} & \rangle \\ \langle C(x,y) = 5 + C(x-1,y) & , & \{x \geq 1, y \geq 0\} & \rangle \end{array}$$

definen la CR C, que tiene dos parámetros y cuyas ecuaciones son válidas para vectores (x, y) tales que $x \geq 0 \wedge y \geq 0$. □

Como ya hemos dicho antes, una característica importante de las CR es su recursión.

Definición 15 (Factor Recursivo de una CR)**.** El **factor recursivo** (recursive factor) b de una CR es el máximo factor recursivo de sus CE. □

Ejemplo 21 (Factor Recursivo de una CR)**.** En el Ejemplo 19, el factor recursivo de la CR C es 2, que es el de la ecuación (3). □

2.3.3. Sistemas de Relaciones de Coste

Definición 16 (Sistemas de Relaciones de Coste)**.** Un **sistema de relaciones de coste** *(cost relation system CRS)* $\mathcal{S}$ es un conjunto de ecuaciones de coste que definen varias CR. Un *CRS* debe ser auto-contenido: sus ecuaciones solo contienen llamadas a otras CR definidas en $\mathcal{S}$. □

La función $rel(\mathcal{S})$ da el conjunto de CR para las que hay una ecuación en $\mathcal{S}$. Para cada CR C, $\mathcal{S}.def(C)$ es el conjunto de ecuaciones de C en $\mathcal{S}$.

Ejemplo 22 (Sistema de Relaciones de Coste)**.** En este *CRS*,

$$\mathcal{S} = \left\{ \begin{array}{llll} \langle C(x,y) & = \mathsf{nat}(x) + 2 & , \ \{x \geq 0, y = 0\} & \rangle \\ \langle C(x,y) & = 1 + D(z) + C(x, y-1) & , \ \{y \geq 1, x \geq 0, 0 \leq z \leq x\} & \rangle \\ \langle D(x) & = 3 & , \ \{x \leq 0\} & \rangle \\ \langle D(x) & = 5 + D(x-1) + E(x-2) & , \ \{x \geq 1\} & \rangle \\ \langle E(x) & = 5 * \mathsf{nat}(x+3) & & \rangle \end{array} \right\}$$

se definen las relaciones $rel(\mathcal{S}) = \{C/2, D/1, E/1\}$. En esta notación C/N C es el nombre de la CR y N es el número de argumentos. □

Ejemplo 23 (*CRS* de Lista.delete)**.** En el *CRS* del método Lista.delete (Figura 1.9) se definen las CR $\{Del/5, C/5, D/3, E/2\}$. □

2.4. Semántica de los *CRS*

En esta Sección explicamos la semántica de una CR dentro de un *CRS*. Para ello distinguimos dos partes: la primera es una especificación formal y no constructiva del conjunto de soluciones de una CR, y la segunda es un algoritmo para construir todas esas soluciones.

2.4.1. Solución de una CR

Una CR de aridad n es una aplicación $C : \mathbb{Z}^n \mapsto \mathcal{P}(\mathbb{R}^+)$. La semántica de la CR debe dar, para cada vector $\bar{v} \in \mathbb{Z}^n$, la especificación del conjunto $C(\bar{v})$. Nosotros damos esa especificación mediante restricción *(restricted comprehension)* de $\mathbb{R}^+$.

Definición 17 (Solución de una CR)**.** Sea $C(\bar{x})$ una CR de aridad n y sea $\bar{v} \in \mathbb{Z}^n$. Un número $r \in \mathbb{R}^+$ es una solución de $C(\bar{v})$, escrito $r \in C(\bar{v})$ si y solo si existe una ecuación $\mathcal{E}$ de la forma

$$\mathcal{E} = \langle C(\bar{x}) = e + \sum_{i=1}^{m} D_i(\bar{y}_i), \varphi \rangle$$

y una asignación $\sigma : \bar{x} \cup \bar{y}_i \cup vars(e) \mapsto \mathbb{R}^+$ tales que $\sigma \models \varphi \wedge \bar{x} = \bar{v}$ y $r = \sigma(e) + \sum_{i=1}^{m} d_i$ y cada d_i cumple $d_i \in D(\bar{w}_i)$, siendo $\bar{w}_i$ el resultado de aplicar σ en cada posición de $\bar{y}_i$. □

Ejemplo 24 (Soluciones de una CR)**.** Sea C la CR definida con estas ecuaciones:

$$\begin{array}{lllll} (1)\langle & C(x) = & 3 & , x \geq 0 & \rangle \\ (2)\langle & C(x) = & 4 + C(x-1) & , x \geq 1 & \rangle \end{array}$$

algunas soluciones de $C(3)$ son las siguientes:

- $3 \in C(3)$ se obtiene al escoger la ecuación (1).
- $7 \in C(3)$ se obtiene si escogemos la ecuación (2) pues $x = 3$ satisface $x \geq 1$ y $7 = 4 + 3$ y podemos ver que $3 \in C(2)$ (Ecuación 1).

Asimismo $11, 15 \in C(3)$ y por tanto $C(3) = \{3, 7, 11, 15\}$. □

Definición 18 (Dominio de una CR)**.** Se define el **dominio** de una CR n-aria C como

$$dom(C) = \{\bar{v} \in \mathbb{Z}^n | C(\bar{v}) \neq \phi\}$$

es decir, el conjunto de vectores enteros de n elementos para los que $C(\bar{v})$ tiene al menos una solución. □

Ejemplo 25 (Dominio de una CR)**.** El dominio de la relación de coste C del Ejemplo 24 es el conjunto de enteros no nulos. □

2.4.2. Evaluación de una CR

La Definición 17 debe complementarse con un algoritmo para evaluar una CR, esto es para generar sus soluciones.

Definición 19 (Evaluación de una CR)**.** Sea $\mathcal{S}$ un *CRS*, C una CR $C \in rel(\mathcal{S})$ y un vector $\bar{v} \in \mathbb{Z}^n$. Una **evaluación** *(evaluation)* de $C(\bar{v})$ es una tupla $T = node(C(\bar{v}), r, \langle T_1, \ldots, T_k \rangle)$ que puede construirse con estos pasos:

1. Escoger una ecuación de coste
$$\mathcal{E} = \left\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi \right\rangle \in \mathcal{S}.def(C)$$
2. Encontrar una asignación $\sigma : vars(\mathcal{E}) \mapsto \mathbb{Z}$ que verifique $\sigma \models \bar{x} = \bar{v} \wedge \varphi$.
3. Calcular el coste local de la evaluación $r = \sigma(e)$.
4. En cada T_i construir una evaluación de $D_i(\bar{w}_i)$ siendo $\bar{w}_i$ el resultado de aplicar σ a cada componente de $\bar{y}_i$. Esas T_i reciben el nombre de evaluaciones derivadas *(child evaluations)* de T.

El resultado de una evaluación T, escrito $\mathsf{Sum}(T)$, es el número definido como
$$\mathsf{Sum}(node(C(\overline{v}), r, \langle T_1, \ldots, T_k \rangle)) = r + \sum_{i=0}^{k} \mathsf{Sum}(T_i)$$
que resulta de acumular los costes locales de esa evaluación. □

Este algoritmo no es determinista porque el paso (1) admite varias soluciones y el paso (2) puede incluso admitir infinitas soluciones. Eso puede provocar que una llamada tenga infinitas evaluaciones.

Ejemplo 26 (Llamada con infinitas evaluaciones)**.** Al evaluar la CR del Ejemplo 2 siguiendo la Definición 19 hay infinitas asignaciones a j que satisfagan $j \geq 0$. Así, $Del(3, 10, 2, 20, 2)$ admite infinitas evaluaciones que generan un conjunto finito de valores. □

Otra característica del algoritmo de evaluación es que algunas *ramas* pueden desembocar en un fallo o ser infinitas. Se dará un fallo si en alguna evaluación derivada no se pueden satisfacer los pasos (1) y (2). Y puede que la evaluación no termine si se genera una reiteración constante del paso (3).

Ejemplo 27 (Evaluación Infinita)**.** Sea C la relación de coste definida por esta única ecuación:
$$\langle C(x) = 1 + C(x) \rangle$$
La evaluación de C nunca termina, y por tanto $\forall x : C(x) = \phi$. □

En lo que respecta a la corrección y completitud del algoritmo, solo interesan las evaluaciones terminantes y exitosas.

Propiedad 3 (Corrección y completitud del algoritmo de evaluación)**.** Respecto a la especificación de soluciones de una CR, (Definición 17) el algoritmo de evaluación (Definición 19) es

- Correcto: toda evaluación T exitosa y terminante de $C(\bar{v})$ satisface que $\mathsf{Sum}(T) \in C(\bar{v})$.
- Completo: para toda solución $r \in c(\bar{v})$ existe una evaluación exitosa y terminante T de $C(\bar{v})$ tal que $r = \mathsf{Sum}(T)$. □

Conceptos asociados a una Evaluación

Definición 20 (Subevaluación)**.** Sean $E_1 =$ y E_2 dos evaluaciones . Se dice que E_2 es una **subevaluación** de índice $k \geq 0$ si sucede que:

- Para $k = 0$, si $E_2 = E_1$.
- Para $k \geq 1$, si hay algún T_i de E_1 tal que E_2 es una subevaluación de índice $k - 1$ de T_i.

Para una evaluación T, el subnivel de índice $k \geq 0$ es el conjunto de subevaluaciones de índice k. La altura de una evaluación T es el máximo índice de un subnivel no vacío de T. □

Para una evaluación T, escribimos $\mathsf{Sum\_Level}(T, i)$ la suma de los costes locales de las subevaluaciones de T que están en el nivel i. Definimos el conjunto de *child local-cost expressions* como el conjunto de tripletas, compuestas cada una por dos expresiones de coste e_1, e_2 enlazadas por las restricciones en que una aplicación de e_1 puede derivar una aplicación de e_2.

Definición 21 (Expresiones de coste locales derivadas)**.** El conjunto de expresiones de coste local derivadas (child local-cost expressions) de una CR independiente C, escrito $Child\_Exps(C)$, se define como

$$Child\_Exps(C) = \left\{ \langle e, e', \psi \rangle \;\middle|\; \begin{array}{l} \langle C(\bar{x}) = e + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle \in \mathcal{S}, \text{ donde } k \geq 1 \\ \forall\ 1 \leq i \leq k.\ \langle C(\bar{y}_i) = e_i + \sum_{j=1}^{k_i} C(\bar{z}_j), \varphi_i \rangle \in \mathcal{S} \\ e' = e_1 + \cdots + e_k \\ \psi = \bar{\exists} vars(e) \cup vars(e').\varphi \wedge \varphi_1 \wedge \cdots \wedge \varphi_k \end{array} \right\}$$

□

2.4.3. Cota Superior e Inferior de una CR

Hemos mencionado en secciones anteriores que queremos obtener cotas superiores e inferiores de relaciones de coste, que sirvan para las aplicaciones prácticas mencionadas en la Sección 1.2. Por eso vamos a definir ahora qué es una cota superior e inferior de una CR.

Definición 22 (Cota superior e inferior de una CR)**.** Sea C una relación de coste de aridad n. Una función $f : \mathbb{Z}^n \mapsto \mathbb{R}^+$ tal que

$$\forall \bar{v} \in dom(C) \forall r \in C(\bar{v}) : f(\bar{v}) \geq r$$

es una **cota superior** *(upper bound, UB)* de C. De manera semejante, f es una **cota inferior** *(lower bound)* si $f(\bar{v}) \leq r$. □

Ejemplo 28 (Cota Superior e Inferior de una CR)**.** La CR del Ejemplo 24 admite las funciones f y g,

$$f(x) = 5x \leftarrow x \geq 0 \qquad\qquad g(x) = x \leftarrow x \geq 0$$

como cota superior e inferior, respectivamente. □

En general, para una CR C emplearemos las notaciones C^+ para denotar una cota superior de C y C^- para denotar una inferior. Toda CR admite muchas cotas superiores o inferiores, pero nos interesa obtener cotas más precisas. La precisión relativa de una cota se observa con respecto a las funciones de valor máximo y mínimo.

Definición 23 (Valor máximo de una CR)**.** Sea C una CR n. La función $max_C : \mathbb{Z}^n \mapsto \mathbb{R}^+$ definida como

$$max_C(\bar{v}) = max(\{r \in C(\bar{v})\})$$

es la **función de valor máximo** *(max value)* de C. Es decir, da la solución más grande (orden usual de $\mathbb{R}^+$) que admite $C(\bar{v})$. □

Ejemplo 29 (Valor máximo de una CR)**.** Las siguientes son funciones de valor máximo y mínimo

$$max_C(x) = 4x + 3 \leftarrow x \geq 0 \qquad\qquad min_C(x) = 3 \leftarrow x \geq 0$$

para la CR del Ejemplo 24. □

Claramente, para una CR C y un valor inicial $\bar{v}$, con frecuencia habrá infinitas maneras de evaluar $C(\bar{v})$ y por ello max_C no será computable. En tales casos tendremos que conformarnos con calcular para $\mathcal{S}$ una cota superior ub menos precisa que max_C . Dicho en términos lógicos, inferimos información siempre correcta y casi siempre imprecisa.

2.5. Contextos y Recursión

En un *CRS* las relaciones de coste estan interconectadas a través de las llamadas externas o recursivas que hay en sus ecuaciones. En el estudio de los *CRS* y en la obtención de resultados en forma cerrada es preciso estudiar de la manera más formal y aislada posible esas interconexiones.

2.5.1. Contextos y bucles

La primera noción sirve para formalizar esa intuición de conexión entre las CR en una fórmula sencilla.

Definición 24 (Contexto de llamada)**.** Un **contexto de llamada** *(call context)* $\mathcal{C}$ es una fórmula $\langle C(\bar{x}) \to D(\bar{y}), \varphi\rangle$ donde C yD son dos CRs y φ es una relación de tamaño entre las variables $\bar{x}, \bar{y}$. C es la CR origen y D es la CR destino. $\mathcal{C}$ es **satisfactible** si lo es su restricción φ. □

Si pensamos en los términos de la Definición 19, $\langle C(\bar{x}) \to D(\bar{y}), \varphi\rangle$ indica que una evaluación de $C(\bar{v}_1)$ puede incluir una subevaluación de $D(\bar{v}_2)$ tales que $\bar{x} = \bar{v}_1 \wedge \bar{y} = \bar{v}_2 \models \varphi$. En otros campos, la noción de contexto podría ser una especificación inexacta de cómo los valores que hay en un nodo C se transforman en las entradas de otro nodo D.

Ejemplo 30 (Contexto simple)**.** Sean C, D dos relaciones de coste monoparamétricas. El contexto

$$\langle C(x) \to D(0), \{x \geq 0\}\rangle$$

me indica que si inicio una evaluación de C con un argumento positivo (o 0), entonces eso puede llevar a que se inicie una evaluación de D con el argumento $y = 0$. □

Este era un ejemplo sencillo, pero veamos ahora uno más realista:

Ejemplo 31 (Contexto de llamada en Lista.delete)**.** En el *CRS* de Lista.delete (Fig. 1.9, Página 11) se induce este contexto de llamada (Ecuación (3)):

$$\langle C(l, a, la, b, lb) \to E(la, j), \{l > 0, a \geq la, a \geq 0, b \geq lb, b \geq 0, j \leq 0\}\rangle$$

que me indica que al evaluar C se puede iniciar una evaluación de E. □

De todas las conexiones que puede haber en un *CRS* las más interesantes son aquellas en que una misma relación es origen y destino.

Definición 25 (Bucle)**.** Un **bucle** *(loop)* de una CR C es un contexto de llamada de C a C. Es decir, es una fórmula

$$\langle C(\bar{x}) \to C(\bar{y}), \varphi \rangle$$

donde φ es una relación de tamaño entre las variables $\bar{x}, \bar{y}$. □

Ejemplo 32 (Bucle Simple)**.** Sea la CR monoparamétrica $C(x)$. El contexto $\langle C(x) \to C(x-1), x \geq 1 \rangle$ es un bucle de C. □

Ejemplo 33 (Bucle en el *CRS* de Lista.delete)**.** En el *CRS* de la Fig. 1.9 (Pág. 11) se induce este bucle en la ecuación (9) de la relación E:

$$\langle E(la, j) \to E(la, j+1), \{0 \leq j < la - 1\}$$

que corresponde al bucle for del método Vector.elimina. □

Un bucle especial, que luego emplearemos en varias definiciones y enunciados, es el bucle nulo de C que corresponde a la transformación que no modifica las variables, esto es el que corresponde a la función identidad.

Definición 26 (Bucle Nulo)**.** Para toda CR C se define el bucle nulo de C como $\mathcal{C}_C^0 = \langle C(\bar{x}) \to C(\bar{y}), \bar{x} = \bar{y} \rangle$. □

En un *CRS* con varias CR puede ocurrir que una relación C cause la evaluación de D (por una llamada directa), y que a su vez D llame a E e inicie su evaluación. Resulta intuitivo unir esas conexiones y así inferir que C puede activar la evaluación de E a través de D. Esta intuición se formaliza en la operación de composición.

Definición 27 (Composición de contextos)**.** La composición de dos contextos simples $\mathcal{C}_1 = \langle C(\bar{x}) \to D(\bar{y}), \varphi_1 \rangle$ y $\mathcal{C}_2 = \langle D(\bar{y}) \to E(\bar{z}), \varphi_2 \rangle$, es el contexto

$$\mathcal{C}_1 \circ \mathcal{C}_2 = \langle C(\bar{x}) \to E(\bar{z}), \bar{\exists}(\bar{x} \cup \bar{z})(\varphi_1 \wedge \varphi_2) \rangle$$

□

Ejemplo 34 (Composición de Contextos)**.** Sean C, D, E tres CR monoparamétricas, y sean los contextos

$$\begin{array}{rcll} I_1 & = & \langle C(x) \to D(x-1) & , \{x \geq 1\} \quad \rangle \\ I_2 & = & \langle D(y) \to E(0) & , \{y \geq 0\} \quad \rangle \end{array}$$

su composición es

$$I_3 = I_1 \circ I_2 = \langle C(x) \to E(0), \{x \geq 1\} \rangle$$

que indica cómo la evaluación de $C(x)$ puede iniciar la de E. □

Propiedad 4 (Propiedades de la Composición $\circ$)**.** Sean C, D, E, F varias CRs y los contextos $\mathcal{C}_1 = \langle C(\bar{x}) \to D(\bar{y}), \varphi_1 \rangle$, $\mathcal{C}_2 = \langle D(\bar{y}) \to E(\bar{z}), \varphi_2 \rangle$ y $\mathcal{C}_3 = \langle E(\bar{z}) \to F(\bar{w}), \varphi_3 \rangle$. La operación de composición $\circ$

1. Es asociativa: $(\mathcal{C}_1 \circ \mathcal{C}_2) \circ \mathcal{C}_3 = \mathcal{C}_1 \circ (\mathcal{C}_2 \circ \mathcal{C}_3)$
2. Tiene elementos neutros por la derecha y la izquierda, que son los bucles nulos de las CRs: $\mathcal{C}^0_C \circ \mathcal{C}_1 = \mathcal{C}_1 \circ \mathcal{C}^0_D = \mathcal{C}_1$. □

Estas propiedades permiten pensar en el conjunto de bucles de una CR como si de una estructura algebraica se tratara. Cuando se estudian las construcciones recursivas de un programa importa conocer el efecto de dar varias iteraciones. En el álgebra de los bucles, el proceso de reiterar un bucle se representa con la operación de potencia.

Definición 28 (Potencia de un Bucle)**.** Sea C una CR y sea $\mathcal{R}$ un bucle de C. Definimos la potencia enésima de $\mathcal{R}$, escrito $\mathcal{R}^n$, con estas ecuaciones:

- La potencia 0 de $\mathcal{R}$ es el bucle nulo de C.
- Para cualquier $n = k + 1$, $\mathcal{R}^{k+1} = \mathcal{R}^k \circ \mathcal{R}$. □

Ejemplo 35 (Potencia de un bucle sencillo)**.** Sea C una CR monoparamétrica y sea I el siguiente bucle de C:

$$I = \langle C(x) \to C(x-1), x \geq 1 \rangle$$

las potencias de I son los siguientes bucles:

$$\begin{array}{rcl} I^0 & = & \langle C(x) \to C(x) \quad , \quad \rangle \\ I^1 & = & \langle C(x) \to C(x-1) \quad , x \geq 1 \quad \rangle \\ I^2 & = & \langle C(x) \to C(x-2) \quad , x \geq 2 \quad \rangle \end{array}$$

y para potencias mayores, $I^n = \langle C(x) \to C(x-n), x \geq n \rangle$. □

Ejemplo 36 (Potencia de Bucle idempotente)**.** Sea C una CR monoparamétrica y sea I el siguiente bucle de C:

$$I = \langle C(x) \to C(0), x \geq 0 \rangle$$

las potencias de I son los siguientes bucles:

$$\begin{array}{rcl} I^0 & = & \langle C(x) \to C(x) \quad , \quad \rangle \\ I^1 & = & \langle C(x) \to C(0) \quad , x \geq 0 \quad \rangle \\ I^2 & = & \langle C(x) \to C(0) \quad , x \geq 0 \quad \rangle \end{array}$$

y como se ve, $\forall n \geq 1 : I^n = I$. Es decir, que el bucle se estabiliza a partir de cierta potencia. □

Ejemplo 37 (Potencias Insatisfactibles de un Bucle)**.** Sea C una CR monoparamétrica y sea I el siguiente bucle de C:

$$I = \langle C(x) \to C(0), x \geq 1 \rangle$$

las potencias de I son los siguientes bucles:

$$\begin{array}{lcl} I^0 & = & \langle C(x) \to C(x) \quad , \qquad \rangle \\ I^1 & = & \langle C(x) \to C(0) \quad , x \geq 1 \quad \rangle \\ I^2 & = & \langle C(x) \to C(0) \quad , False \quad \rangle \end{array}$$

y como se ve cualquier potencia I^n con $n \geq 2$ es insatisfactible. Intuitivamente, representa un bucle que da a lo sumo una iteración. □

Ejemplo 38 (Potencia de Bucles de Lista.delete)**.** Vamos a calcular algunas potencias del bucle múltiple $\mathcal{C}$ que contiene los bucles simples obtenidos en el Ejemplo 53 con los valores $\bar{x}_0 = \langle l_0, la_0, lb_0 \rangle$ y $\bar{x} = \langle l, la, lb \rangle$.

$$\begin{array}{l} I_1 = \langle C(l, la, lb) \to \quad C(l', la - 1, lb), \{l > l' \geq 0\} \rangle \\ I_2 = \langle C(l, la, lb) \to \quad C(l', la, lb - 1), \{l > l' \geq 0\} \rangle \end{array}$$

- la potencia nula $\mathcal{C}^0$ es el bucle nulo $\langle C(\bar{x}_0) \to C(\bar{x}), \bar{x}_0 = \bar{x} \rangle$.
- Tras una iteración, $\mathcal{C}^1 = \mathcal{C} = \{I_1, I_2\}$.

  $$\begin{array}{l} I_1 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{\underline{l < l_0}, \underline{la = la_0 - 1}, lb = lb_0, \underline{l_0 > 0}\} \rangle \\ I_2 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{\underline{l < l_0}, la = la_0, \underline{lb = lb_0 - 1}, \underline{l_0 > 0}\} \rangle \end{array}$$

  Se subrayan las restricciones modificadas por aplicar el bucle.
- Tras dos iteraciones, $\mathcal{C}^2 = \{I_3, I_4, I_5, I_6\}$

  $$\begin{array}{l} I_3 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, \underline{la = la_0 - 2}, lb = lb_0, l_0 > 0\} \rangle \\ I_4 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, la = la_0, \underline{lb = lb_0 - 2}, l_0 > 0\} \rangle \\ I_5 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, la = la_0 - 1, \underline{lb = lb_0 - 1}, l_0 > 0\} \rangle \\ I_6 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, \underline{la = la_0 - 1}, lb = lb_0 - 1, l_0 > 0\} \rangle \end{array}$$

  donde $I_3 = I_1^2$, $I_4 = I_2^2$,$I_5 = I_1 \circ I_2$ e $I_6 = I_2 \circ I_1$. La alteración de las restricciones refleja cómo, en cada iteración, disminuye la o lb.
- A partir de ahí, la enésima potencia contiene todos los bucles de la forma $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, \underline{la = la_0 - na}, lb = lb_0 - nb, l_0 > 0\} \rangle$ donde $na \leq 0, nb \leq 0$ y $n = na + nb$. □

Por último, vamos a definir las cerraduras de los bucles:

Definición 29 (Cerraduras de bucles)**.** Sea L un bucle, se definen las cerraduras de L como $L^+ = \vee_{i=1}^{n} L^n$ y $L^* = \vee_{i=0}^{n} L^n\}$.

Ejemplo 39 (Cerraduras de Bucles)**.** Las cerraduras del bucle del Ej. 35. son $I^+ = \langle C(x) \to C(y), x > y \geq 0 \rangle$, $I^* = \langle C(x) \to C(y), x \geq y \geq 0 \rangle$. □

2.5.2. Sistema de Bucles

La mayoría de los problemas en análisis de programas vienen del estudio de sus construcciones iterativas, las cuales corresponden a las instrucciones while o for en un lenguaje imperativo o a las llamadas recursivas en uno declarativo. Para estudiar esas construcciones se deben formalizar las circunstancias en que se inicia, se continúa o termina su ejecución. Para ello nosotros empleamos la noción del sistema de bucles, que aquí presentamos.

Precondiciones y Postcondiciones

Cuando se habla de la ejecución de un programa, es usual formalizar las circunstancias en que éste puede ejecutarse y en las cuales puede terminar.

Definición 30 (Precondición Simple)**.** Sea C una CR de parámetros $\bar{x}$. Una precondición simple de C es una fórmula $\mathcal{P} = \langle \top \rightarrow C(\bar{x}), \varphi \rangle$ donde φ es una relación de tamaño entre las variables $\bar{x}$. □

Intuitivamente, una precondición me dice que es posible iniciar una evaluación o llamada a C desde el entorno (representado con $\top$) si los parámetros satisfacen la restricción φ.

Ejemplo 40 (Precondición Simple)**.** Sea $C(x)$ una CR. Una posible precondición es $\langle \top \rightarrow C(x), \{x > 0\} \rangle$ es decir, C admite valores positivos. □

Ejemplo 41 (Precondición en Lista.delete)**.** En la Fig.1.9 una precondición de la CR Del es

$$\langle \top \rightarrow Del(l, a, la, b, lb), \{l \geq 0, a \geq la \geq 0, b \geq lb \geq 0\} \rangle$$

que es la restricción de tamaño de su única ecuación. □

Definición 31 (Postcondición Simple)**.** Sea C una CR de parámetros $\bar{x}$. Una postcondición simple de C es una fórmula $\mathcal{O} = \langle C(\bar{x}) \rightarrow \bot, \varphi \rangle$ donde φ es una relación de tamaño entre las variables $\bar{x}$. □

Intuitivamente, una postcondición indica que la evaluación de C puede acabar cuando los parámetros satisfacen la restricción φ.

Ejemplo 42 (Postcondición Simple)**.** Sea $C(x)$ una CR. Una posible postcondición es $\langle C(0) \rightarrow \bot \rangle$ es decir, C puede terminar al llegar a 0. □

Ejemplo 43 (Postcondición en Lista.delete)**.** En la Fig.1.9 una postcondición de la CR E es $\langle E(la, j) \rightarrow \bot, \{j \geq la - 1, j \geq 0\} \rangle$, que es justamente la restricción de tamaño de la ecuación (8), que es el caso base de E. □

Sistema de Bucles

En el bucle de un programa iterativo se suelen especificar una serie de precondiciones, que se cumplen a la entrada del bucle, postcondiciones (que se cumplen a la salida), así como la condición iterativa (guarda).

En el análisis de coste, estos elementos se interpretan en un sistema de bucles, que agrega precondiciones, postcondiciones y bucles.

Definición 32 (Sistema de Bucles)**.** Sea C una CR de aridad n. Un Sistema de Bucles de C, o Sistema Recursivo, es una tupla $\langle \mathcal{P}, \mathcal{R}, \mathcal{O} \rangle$ donde

- $\mathcal{P}$ son varias precondiciones de C.
- $\mathcal{R}$ son varios bucles de C y
- $\mathcal{O}$ son varias postcondiciones de C. □

Los sistemas de bucles permiten modelar el comportamiento de las construcciones iterativa, como muestra el siguiente Ejemplo.

Ejemplo 44 (Ejemplo de un Sistema de Bucles)**.** Considérese el clásico programa iterativo que calcula el factorial de un número, a la izquierda de la Figura 2.1. El comportamiento de la instrucción while de este programa se modela con el sistema de bucles que está a la derecha: la variable n empieza con cualquier valor, mientras tenga un valor positivo da otra iteración en que su valor se decrementa hasta que tiene un valor menor o igual que 0. □

```
int factorial(int num){
    int fact = 1;
    for(int n=num ; n>0;n--){
        fact *= n
    } //Salida: n=0
    return fact;
}
```

$$\left\{ \begin{array}{lcccl} \langle & \top & \rightarrow & C(x) & \rangle \\ \langle & C(x) & \rightarrow & C(x-1) \quad , x \geq 1 & \rangle \\ \langle & C(x) & \rightarrow & \bot \quad , x \leq 0 & \rangle \end{array} \right\}$$

Figura 2.1: Programa Factorial y Sistema de Bucles Asociado

Ejemplo 45 (Sistema de Bucles en Lista.delete)**.** En la Fig.1.9 se puede especificar el comportamiento recursivo de la CR $E(la, j)$ con este sistema de bucles:

$$\left\{ \begin{array}{lcccl} \langle & \top & \rightarrow & E(la,j) & , \{j \geq 0\} & \rangle \\ \langle & E(la,j) & \rightarrow & E(la,j+1) & , \{j < la-1, j \geq 0\} & \rangle \\ \langle & E(la,j) & \rightarrow & \bot & , \{j \geq la-1, j \geq 0\} & \rangle \end{array} \right\}$$

siendo la postcondición la mostrada en el Ej 43. □

Número de Iteraciones

Uno de los mayores desafíos al estudiar y verificar un programa iterativo es el de saber cuántas iteraciones dará como máximo (o como mínimo) el bucle antes de terminar; o incluso, si éste terminará. Dado que nosotros empleamos los sistemas de bucles como la abstracción de esos programas iterativos, debemos reflejar en ellos dichos conceptos.

El primero es el de la cota superior del número de iteraciones, que limita cuántas iteraciones puede llegar a dar un bucle.

Definición 33 (Cota Superior del número de iteraciones)**.** Sea $\mathcal{L} = \langle \mathcal{P}, \mathcal{R}, \mathcal{O} \rangle$ un Sistema de Bucles. El número $n \in \mathbb{N}$ es una **cota superior** del número de iteraciones de $\mathcal{L}$ si

$$\forall m > n : \mathcal{P} \circ \mathcal{R}^m \text{ es insatisfactible}$$

El sistema $\mathcal{L}$ es **terminante** si admite una cota superior del número de iteraciones. □

Ejemplo 46 (Cotas del número de iteraciones)**.** En el sistema de bucles del Ejemplo 44, para cualquier valor inicial x_0 la fórmula $\mathsf{nat}(x_0)$ proporcionan una cota superior e inferior, esto es el valor exacto, del número de iteraciones de ese bucle. Por tanto, se trata de un sistema de bucles terminante. □

Ejemplo 47 (Cotas del número de iteraciones en Lista.delete)**.** El sistema de bucles del Ejemplo 45 admite como cota superior e inferior del número de iteraciones la fórmula $\mathsf{nat}(la - j)$. □

De manera análoga, el concepto de cota inferior me indica que un bucle no termina antes de dar n iteraciones.

Definición 34 (Cota Inferior del Número de Iteraciones)**.** Sea $\mathcal{L} = \langle \mathcal{P}, \mathcal{R}, \mathcal{O} \rangle$ un Sistema de Bucles. Si el número $n \in \mathbb{N}$ cumple

$$\forall m > n : \mathcal{P} \circ \mathcal{R}^m \text{ es insatisfactible}$$

entonces es una **cota inferior** del número de iteraciones de $\mathcal{L}$. □

En los ejemplos anteriores, se da el caso de que las cotas superiores del número de iteraciones también son cotas inferiores.

Invariantes

Otro de los aspectos fundamentales del comportamiento de un programa iterativo es saber qué valen sus variables en cada iteración. Como generar cada una de ellas es un proceso no terminante e intratable; resulta más sencillo buscar una fórmula que se cumpla al comenzar la ejecución de ese programa y durante todas sus iteraciones, lo que se conoce como un invariante.

Definición 35 (Invariante de un Sistema de Bucles)**.** Sea $\mathcal{L} = \langle \mathcal{P}, \mathcal{R}, \mathcal{O} \rangle$ un Sistema de Bucles de C. Un **invariante** *(invariant)* de $\mathcal{L}$ es un bucle $\mathcal{I}$ t.q.

$$\forall n \in \mathbb{N} : \mathcal{P} \circ \mathcal{R}^n \models \mathcal{I}$$

esto es, si se cumple al entrar y durante todas sus iteraciones. □

Ejemplo 48 (Invariante de un sistema de bucles.)**.** El sistema del Ejemplo 44, admite como invariante el bucle simple $\langle C(x_0) \to C(x), 0 \leq x \rangle$

Es evidente que si $\mathcal{C}$ es un invariante del sistema $\langle \mathcal{P}, \mathcal{R}, \mathcal{O} \rangle$, entonces cualquier $\mathcal{C}'$ más débil que $\mathcal{C}$ también es un invariante correcto. En particular, todo sistema de bucles admite el invariante trivial $\langle C(\overline{x}) \to C(\overline{y}), \top \rangle$. Al comparar la precisión de los invariantes, se toma como referencia el exacto.

Definición 36 (Invariante Exacto de un Sistema de Bucles)**.** El bucle múltiple $\mathcal{I}$ definido como

$$\mathcal{I} = \mathcal{P} \circ \mathcal{R}^*$$

es el invariante exacto del sistema de bucles. Cualquier otro invariante $\mathcal{I}'$ debe satisfacer $\mathcal{I} \models \mathcal{I}'$. □

Ejemplo 49. El bucle $\langle C(x_0) \to C(x), 0 \leq x \leq x_0 \rangle$ es el invariante exacto del sistema de bucles del ejemplo 44. □

2.5.3. Contextos y dependencias en un *CRS*

Las definiciones han formalizado los conceptos de contexto de llamada, bucle y sistema de bucles. Ahora el el interés es emplearlos en el estudio de los *CRS* para observar cómo se conectan dos CR o cuáles son los ciclos de recursividad de cada una.

Contextos inducidos por un *CRS*

Para empezar a analizar esas conexiones dentro de un *CRS*, comenzamos por la noción de contexto directo inducido.

Definición 37 (Contexto Directo Inducido)**.** Sea $\mathcal{E}$ una CE

$$\mathcal{E} = \langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi \rangle$$

Para cada CR D, el conjunto $\mathcal{E}.Contexts(D)$ definido como

$$\mathcal{E}.Contexts(D) = \{\langle C(\bar{x}) \to D(\bar{y}_i), \bar{\exists}\bar{x} \cup \bar{y}_i.\varphi \rangle | D = D_i\}$$

es el conjunto de *contextos* inducido por $\mathcal{E}$. □

Ejemplo 50 (Contexto Directo Inducido)**.** La siguiente ecuación de coste

$$\langle C(x) = \mathsf{nat}(x) + D(x-1, z) + E(y), \{x \geq 1, x \leq z \leq y \leq 2x\} \rangle$$

induce los siguientes contextos

$$\begin{array}{llll} \langle C(x) \to & D(x-1,z) & , & \{x \geq 1, x \leq z\} \quad \rangle \\ \langle C(x) \to & E(y) & , & \{x \geq 1, x \leq y \leq 2x\} \quad \rangle \end{array}$$

cada uno de los cuales corresponde a una llamada en la ecuación. □

Ejemplo 51 (Contexto Directo Inducido en Lista.delete)**.** Sea $\mathcal{E}$ la ecuación (1) del *CRS* de la Fig. 1.9, se puede ver que

$$\begin{array}{rl} \mathcal{E}_1.Contexts(Del) = & \langle Del(l,a,la,b,lb) \to C(l,a,la,b,lb) \\ & , \{l \geq 0, a \geq la, la \geq 0, b \geq lb, b \geq 0\} \rangle \end{array}$$

es el único contexto en $\mathcal{E}.Contexts(C)$. Por su parte, la ecuación (3), podemos ver que induce estos contextos

$$\begin{array}{rll} \mathcal{E}_3.Contexts(D) = & \langle C(l,a,la,b,lb) \to D(a,la,0) & \\ , & \{l > 0, a \geq la, a \geq 0, b \geq lb, b \geq 0, \} & \rangle \\ \mathcal{E}_3.Contexts(E) = & \langle C(l,a,la,b,lb) \to E(la,j) & \\ , & \{l > 0, a \geq la, a \geq 0, b \geq lb, b \geq 0, j \geq 0\} & \rangle \end{array}$$

para las relaciones de coste D y E. □

Definición 38 (Contextos directos inducidos entre CR por *CRS*)**.** Sea $\mathcal{S}$ un *CRS* y $C, D \in rel(\mathcal{S})$. El conjunto $\mathcal{S}.Contexts(C, D)$, definido como

$$\mathcal{S}.Contexts(C, D) = \bigcup_{\mathcal{E} \in def(\mathcal{S}, C)} \mathcal{E}.Contexts(D)$$

es el conjunto de contextos directos entre C y D inducido por $\mathcal{S}$. □

Ejemplo 52 (Contextos directos inducidos entre CR por *CRS*)**.** En el *CRS* de la Figura 1.9 se induce este contexto directo

$$\langle \quad Del(l, a, la, b, lb) \quad \rightarrow \quad C(l, a, la, b, lb) \quad , \quad \{l \geq 0, a \geq la, la \geq 0, b \geq lb, b \geq 0\} \quad \rangle$$

entre las relaciones Del y C. □

Definición 39 (Bucles Inducidos)**.** Para un *CRS* $\mathcal{S}$ y una CR C, el conjunto definido como

$$\mathcal{S}.Loops(C) = \mathcal{S}.Contexts(C, C)$$

es el conjunto de **bucles** inducido por $\mathcal{S}$ se: □

Ejemplo 53. La CR en la Fig. 1.9 induce estos bucles

$$\langle C(l, a, la, b, lb) \rightarrow C(l', a, la - 1, b, lb), \{a \geq 0, b \geq 0, a \geq la, b \geq lb, l > l', l > 0\}\rangle$$
$$\langle C(l, a, la, b, lb) \rightarrow C(l', a, la, b, lb - 1), \{a \geq 0, b \geq 0, a \geq la, b \geq lb, l > l', l > 0\}\rangle$$

para la relación C. □

Llegamos ahora a una de las definiciones más importantes, porque es la base para estudiar la recursividad en los *CRS* y para definir algunas transformaciones del cálculo de cotas superiores. Intuitivamente, la idea es componer los contextos directos inducidos por el *CRS*.

Definición 40 (Contextos Indirectos Inducidos)**.** Sea un *CRS* $\mathcal{S}$. El conjunto de contextos indirectos inducidos por $\mathcal{S}$ ($\mathcal{S}.Contexts^+$) es el mínimo conjunto que

- Incluye todos los conjuntos inducidos directos:

$$\forall C, D \in rel(\mathcal{S}) : Contexts(C, D) \subseteq Contexts^+(\mathcal{S})$$

- Y es cerrado bajo la operación de composición de contextos.

Sobre $\mathcal{S}.Contexts^+$ se definen

$$\mathcal{S}.Contexts^+(C, D) = \left\{\langle C(\bar{x} \rightarrow D(\bar{y}), \varphi\rangle \in \mathcal{S}.Contexts^+\right\}$$

el conjunto de contextos indirectos entre dos CR y

$$\mathcal{S}.Loops^+(C) = \left\{\langle C(\bar{x} \rightarrow C(\bar{y}), \varphi\rangle \in \mathcal{S}.Contexts^+\right\}$$

el conjunto de bucles indirectos de una CR. □

Dependencias

La definición de $Contexts^+$ sirve para describir de manera nítida el tipo de interconexión y acoplamiento que existe entre las CR de un *CRS*, así como la recursividad de cada una de ellas.

El primer tipo de acoplamiento es la dependencia. En programación, un módulo A depende de otro módulo B si es necesario utilizar B para poder usar A. Del mismo modo, una CR C depende de otra si al evaluar C es posible que necesitemos evaluar D.

Definición 41 (Dependencias entre CRs)**.** Sea un *CRS* $\mathcal{S}$ y $C, D \in rel(\mathcal{S})$. Existe una **dependencia** de C a D (C depende de D) si sucede que

$$\mathcal{S}.depende(C, D) \Leftrightarrow \exists \langle C(\bar{x}) \rightarrow D(\bar{y}), \varphi \rangle \in \mathcal{S}.contexts^+ \wedge \varphi \neq False$$

es decir, $\mathcal{S}$ induce un contexto indirecto satisfactible entre C y D. Si C solo depende de si misma se dice que es es **independiente** *(stand alone)*. □

Ejemplo 54 (Dependencias en un *CRS* sencillo)**.** Vamos a observar un sencillo *CRS* de ejemplo para observar los conceptos que estamos definiendo. En el *CRS* de la Figura 2.2 hay cinco CR con estas dependencias: G solo tiene una ecuación sin llamadas, luego es independiente. F tiene un caso base y una ecuación recursiva, y ninguna contiene llamadas externas. Es CR independiente. El caso recursivo de E contiene una llamada a F, luego E depende de F. Por último, en las relaciones C, D. se ve que C llama a D y viceversa, por lo que son interdependientes. □

$$
\begin{array}{llll}
\langle & C(x) = 1 + D(x) & & \rangle \\
\langle & D(x) = 1 & & \rangle \\
\langle & D(x) = 1 + D(x-1) + C(x-1) & , \quad x \geq 1 & \rangle \\
\langle & E(x) = 1 & & \rangle \\
\langle & E(x) = 1 + E(x-1) + F(x) + G(x) & , \quad x \geq 1 & \rangle \\
\langle & F(x) = 1 & & \rangle \\
\langle & F(x) = 1 + F(x-1) & , \quad x \geq 1 & \rangle \\
\langle & G(x) = 1 & & \rangle
\end{array}
$$

Figura 2.2: *CRS* de ejemplo de recursividad y dependencias

Ejemplo 55 (Dependencias en Lista.delete)**.** En el *CRS* de la Fig. 1.9 se dan las siguientes dependencias: D, E son independientes, C depende de D, E y Del depende de C, y por tanto también depende de D, E. □

2.5.4. Recursividad

Una vez que hemos definido el tipo de conexión y dependencia que puede haber entre dos CR distintas, vamos a discutir cuál es el acoplamiento de una CR sobre sí misma.

Definición 42 (Recursividad entre CRs)**.** Sea un *CRS* $\mathcal{S}$ y sean $C \in rel(\mathcal{S})$. C es recursiva si depende de sí misma, es decir

$$\mathcal{S}.esRecursiva(C) \Leftrightarrow \exists \langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi \rangle \in \mathcal{S}.contexts^{+} \wedge \varphi \neq False$$

esto es si existe un contexto satisfactible no nulo de C a C. □

Definición 43 (Recursividad directa)**.** Sea un *CRS* $\mathcal{S}$ y sean $C \in rel(\mathcal{S})$. C es **directamente recursiva** si para toda CR D tal que C depende de D se cumple que D no depende de C.

En términos de contextos, si $\mathcal{S}.Loops^{+}(C) = (\mathcal{S}.Loops(C))^{+}$ □

Podemos observar que una CR independiente siempre es directamente recursiva o no recursiva.

Ejemplo 56 (Recursividad en un *CRS* sencillo)**.** Las recursividades en el *CRS* de la Figura 2.2 son, de abajo hacia arriba:

- La relación G solo tiene una ecuación sin llamadas. Claramente, G es no recursiva.
- F tiene un caso base y una ecuación recursiva, y no contiene llamadas externas, luego es directamente recursiva.
- E tiene un caso base y un caso recursivo, y en éste hay una llamada a F pero como F no depende de E, se concluye que E es directamente recursiva.
- Por último, observemos las relaciones C, D. Vemos que C llama a D y viceversa, por lo que son CRs indirectamente recursivas. □

Hay que señalar cómo la directa o indirecta recursividad de una CR no solo depende de si sus ecuaciones contienen llamadas recursivas. También se debe observar si las llamadas externas forman parte de un bucle indirecto.

Ejemplo 57 (Recursividad de CR en Lista.delete)**.** En el *CRS* de la Fig. 1.9 las relaciones C, D, E son directamente recursivas, y la relación Del es no recursiva. En este *CRS* no existen recursividades indirectas porque ya se ha aplicado la evaluación parcial (Sección 3.3). □

Definición 44 (CRS directamente recursivo)**.** Un *CRS* está en forma **directamente recursiva** si todas las CRs en $rel(\mathcal{S})$ son no recursivas o directamente recursivas. □

Ejemplo 58 (CRS directamente recursivo)**.** El *CRS*, como contiene una CR indirectamente recursiva, no está en forma directamente recursiva. Puede pasar a estarlo si se despliega, por ejemplo, la CR C, tras lo que se obtiene el *CRS* de la Figura 2.3. □

$$
\begin{array}{llllll}
\langle & C(x) = & 1 & +D(x) & & \rangle \\
\langle & D(x) = & 1 & & & \rangle \\
\langle & D(x) = & 2 & +D(x-1)+D(x-1) & , \quad x \geq 1 & \rangle \\
\langle & E(x) = & 1 & & & \rangle \\
\langle & E(x) = & 1 & +E(x-1)+F(x)+G(x) & , \quad x \geq 1 & \rangle \\
\langle & F(x) = & 1 & & & \rangle \\
\langle & F(x) = & 1 & +F(x-1) & , \quad x \geq 1 & \rangle \\
\langle & G(x) = & 1 & & & \rangle
\end{array}
$$

Figura 2.3: *CRS* de la Figura 2.2, transformado a forma directamente recursiva.

Grafo de llamadas

Los conceptos de dependencia y recursividad son fundamentales en el análisis de *CRSs* y para el cálculo de resultados en forma cerrada. Pero éstos se basan en el conjunto de contextos indirectos inducidos por el *CRS* (Definición. 41) y normalmente calcular ese conjunto es imposible o intratable. Para sortear este obstáculo, aproximamos ese conjunto de contextos mediante el grafo de llamadas del *CRS*, que siempre es computable.

Definición 45. Sea $\mathcal{S}$ un *CRS*. El **grafo de llamadas** *(call graph)* de $\mathcal{S}$ es un grafo dirigido con bucles $\langle V, E \rangle$ tal que

- El conjunto de nodos es $V = rel(\mathcal{S})$.
- Para cada dos nodos $v, w \in V$, que corresponden a dos CRs $C, D \in rel(\mathcal{S})$, hay una arista $(v, w) \in E$ si y solo si C llama a D.

□

Propiedad 5 (Dependencias y grafo de llamadas)**.** Sea $\mathcal{S}$ un *CRS* y su grafo de llamadas $\mathcal{G}(\mathcal{S})$, y dos CRs $C, D \in rel(\mathcal{S})$. Si C depende de D, entonces debe existir un camino en el grafo desde el vértice de C al vértice de D. □

Demostración. Para que C dependa de D, $\mathcal{S}$ debe inducir un contexto $\langle C(\bar{x}) \to D(\bar{y}), \varphi \rangle$ en y todo contexto $I \in contexts^+(\mathcal{S})$ surge como la composición de $n \geq 0$ contextos directos $I = I_1 \circ I_2 \circ \cdots \circ I_n$ y cada contexto directo I_i corresponde a una llamada en una ecuación de coste. Por tanto, cada contexto indirecto corresponde a una cadena de llamada entre ecuaciones de coste, lo que corresponde a un camino en el grafo de llamadas. □

El recíproco no siempre se cumple como muestra el siguiente contraejemplo:

Ejemplo 59 (Grafo de llamadas - Dependencias)**.** En el siguiente ejemplo, podemos ver que hay llamadas de C a D y una de D a E, por lo que en el grafo de llamadas habrá un camino desde C a E.

$$\begin{array}{lll} \langle C(x) = & 1 + D(x) & , x \geq 0 \ \rangle \\ \langle D(x) = & 2, & , x \geq 0 \ \rangle \\ \langle D(x) = & 2 + E(x) & , x < 0 \ \rangle \\ \langle E(x) = & 0 & \rangle \end{array}$$

Sin embargo, si analizamos las guardas de cada ecuación, vemos que no hay un contexto satisfactible desde C a E, y por tanto C no depende de E. □

Dicho en términos lógicos, el grafo de llamadas de un *CRS* $\mathcal{S}$ proporciona un modelo completo pero no siempre correcto del conjunto de contextos indirectos inducidos $\mathcal{S}.Context^+$.

Propiedad 6 (Grafo directamente recursivo)**.** Sea $\mathcal{S}$ un *CRS* directamente recursivo, sea $\mathcal{G}(\mathcal{S})$ su grafo de llamadas y sea $\overline{\mathcal{G}(\mathcal{S})}$ el resultado de suprimir en $\mathcal{G}(\mathcal{S})$ todos los bucles (aristas de un vértice a sí mismo). Se cumple entonces que $\overline{\mathcal{G}(\mathcal{S})}$ es un grafo acíclico. □

Definición 46 (Orden topológico de un grafo)**.** Sea $G = \langle V, E \rangle$ un grafo dirigido acíclico. Existe un orden total $\leq$ entre sus vértices tal que

$$\forall v, w \in V : (v, w) \in E \to v \leq w$$

Tal ordenación recibe el nombre de **orden topológico** de G.

Ejemplo 60 (Órden topológico de un Grafo de Llamadas)**.** Sea $\overline{\mathcal{G}(\mathcal{S})}$ el grafo de llamadas sin bucles del *CRS* $\mathcal{S}$ de la Figura 2.3. Las siguientes secuencias

$$\begin{array}{ll} (C, D, E, F, G) & (E, F, G, C, D) \\ (C, D, E, G, F) & (C, E, F, G, , D) \end{array}$$

denotan órdenes topológicos de ese grafo. □

Capítulo 3

Cálculo de Cotas Superiores

En la Sección 2.4.3 se definió el concepto de cota superior o inferior de una relación de coste, pero no se presentó ningún algoritmo para intentar calcular una. En este capítulo vamos a recordar el método presentado por Albert, Arenas, Genaim y Puebla en [10] para obtener expresiones de coste que representan cotas superiores de relaciones de coste.

3.1. Composicionalidad

Para implementar la resolución de las CR de un *CRS* vamos a emplear el esquema voraz. Para ello debe haber alguna incrementalidad en los datos, es decir debemos buscar una manera de resolver el *CRS* tratando sus CR una a una. El principio de **composicionalidad** *(compositionality)* de los *CRS* nos permite procesar las CRs una por una.

Propiedad 7 (Composicionalidad de Cotas Superiores)**.** Sea un *CRS* $\mathcal{S}$, una CR C y sea $\mathcal{E}$ la ecuación

$$\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$$

donde cada $D_i(\bar{y}_i)$ es otra relación de coste definida en $\mathcal{S}$ que no depende de C, y que admite una cota superior D_i^+ tal que $vars(D_i^+) \subseteq \bar{y}_i$. Si modificamos la ecuación anterior por esta

$$\langle C'(\bar{x}) = e + \sum_{i=1}^{k} D_i^+ + \sum_{j=1}^{n} C'(\bar{z}_j), \varphi \rangle$$

cualquier cota superior de la nueva C también lo es de la original. □

Demostración. Se basa en la monotonía de + (Propiedad 2). □

Ejemplo 61 (Composicionalidad de Cotas Superiores)**.** Si tenemos las cotas superiores $D^+(a, la, i) = 8{+}10{*}\mathsf{nat}(la{-}i)$ y $E^+(la, j) = 5{+}15{*}\mathsf{nat}(la{-}j{-}1)$ entonces al reemplazar las llamadas a D y E en las Ecuaciones (3,4) de la Figura 1.9 por D^+, E^+ se obtiene el CR mostrado en la Fig. 3.1. □

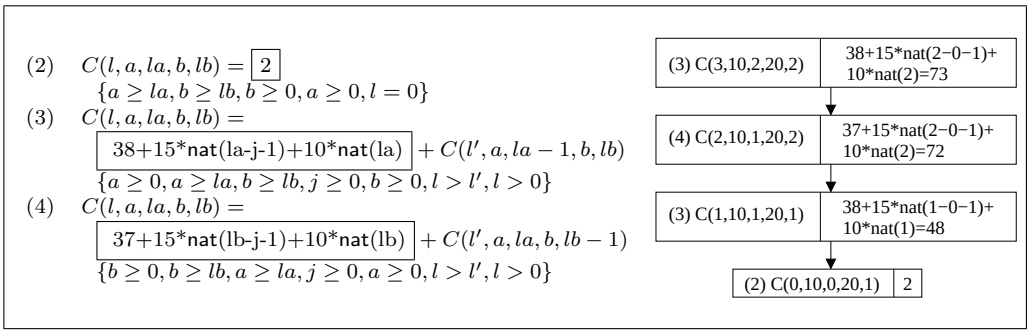

Figura 3.1: CR independiente, junto con una evaluación.

La mejor forma de explotar el principio de composicionalidad es en un *CRS* directamente recursivo. Hemos mencionado (Subsección 2.5.4) que el grafo de llamadas de éstos *CRS* es acíclico (salvo bucles), y por tanto existe un orden topológico entre sus vértices. Nuestro algoritmo (Figura 3.2) recorre las CR siguiendo en orden ascendente ese orden topológico, y para cada CR D calcula su cota superior D^+ y la despliega en todas las llamadas a D.

Para aplicar este algoritmo necesitamos convertir un *CRS* a forma directamente recursiva, usando la *Evaluación Parcial* [35], (Sección 3.3). Para calcular una cota de una CR independiente directamente recursiva, usamos dos fórmulas alternativas: la de Acumulación de Subevaluaciones (Sección 3.2) y la método de Acumulación por Niveles (Sección 3.6).

Require: $\mathcal{S}$ está en forma directamente recursiva
1: **function** RESUELVE($\mathcal{S}$)
2: Tabla[CR,CExp] ubs = tablaVacía()
3: $lista = \mathcal{S}.daOrdenTopologico()$
4: **for** $C : lista$ **do**
5: $C^+ = C.resuelveCotaSuperior()$
6: $ubs.put(C, C^+)$
7: $\mathcal{S}.despliegaLlamadas(C, C^+)$
return ubs

Figura 3.2: Algoritmo de Resolución de un *CRS* directamente recursivo

3.2. Acumulación de Subevaluaciones

Los enfoques para resolver recurrencias describen el conjunto de posibles evaluaciones de cada recurrencia en términos de cantidad de iteraciones, número de veces que se usa cada ecuación, costes locales, etcétera. Vamos a extender ese esquema de aproximación para calcular cotas superiores de los posibles valores que admite una llamada. Su idea central es acumular, o acotar superiormente, el número de subevaluaciones básicas y recursivas y multiplicar esos resultados, respectivamente, por una cota superior del coste local de cada subevaluación básica o recursiva. La cota superior que así se obtiene es correcta aun cuando esa evaluación pesimista -estimación pesimista de subevaluaciones con estimación pesimista del coste en cada una- no sea posible en el CR. Aun cuando no sea posible una evaluación donde se junten el caso pésimo del número de subevaluaciones con el coste local pésimo, el resultado así obtenido es correcto.

Ejemplo 62 (Valor máximo de una CR)**.** Observemos la evaluación de la relación de coste C representada en la Figura 3.1, hay tres subevaluaciones y una básica. Los valores locales de las primeras son 73, 72, y 48, siendo 73 la máxima. El único valor básico posible es 2. Y así, el valor máximo de C es $3 \times 73 + 2 = 221$. Podemos comprobar que efectivamente $221 \geq 73 + 72 + 48 + 2 = 193$. □

Proposición 3 (Cota Superior basada en Acumulación de Subevaluaciones)**.** Sea C una CR independiente, compuesta de nb casos base $\langle C(\bar{x}){=}base_j, \varphi_j\rangle$ para $1{\leq}j{\leq}nb$, y nr ecuaciones recursivas $\langle C(\bar{x}){=}rec_j{+}\sum_{i=1}^{k_j} C(\bar{y}_i), \varphi_j\rangle$ con $1{\leq}j{\leq}nr$. La expresión de coste $C^+(\bar{x})$ definida como

$$C^+(\bar{x}) = \mathsf{l_r} * costr^+(\bar{x}) + \mathsf{l_b} * costnr^+(\bar{x}) \tag{3.1}$$

es una cota superior de C si $\mathsf{l_b}$, $costr^+$, $\mathsf{l_r}$, $costnr^+$ son expresiones de coste tales que en cualquier evaluación de $C(\bar{v})$

1. El número de subevaluaciones es menor que $\mathsf{l_r}$,
2. hay a lo sumo $\mathsf{l_b}$ subevaluaciones básicas,
3. los costes locales de una subevaluación recursiva no supera $costr^+(\bar{v})$ y
4. $costnr^+(\bar{v})$ acota el coste local de toda subevaluación básica. □

Demostración. La proposición es trivialmente correcta. □

Para calcular $\mathsf{l_r}$ e $\mathsf{l_b}$ vamos a diseñar la evaluación pesimista de C, que se da cuando cada evaluación local de C provocara b evaluaciones derivadas, siendo b el factor recursivo de C. En ese caso, la evaluación pésima se asemeja a un árbol completo de ramificación b como el que se muestra en la Figura 3.3. Hay una fórmula combinatoria para determinar el número de nodos hoja (casos base) y no hoja (recursivos) a partir del factor de ramificación y la altura del árbol, lo que para nosotros son el factor recursivo de C y una cota superior $h^+(\bar{x})$ del número de iteraciones de C.

$$\mathsf{l_r} = \begin{cases} \frac{1}{\mathsf{nr}-1}\mathsf{nr}^{h+}(\bar{x}) & \mathsf{nr} > 1 \\ h^+(\bar{x}) & \text{en otro caso} \end{cases} \qquad \mathsf{l_b} = \begin{cases} \mathsf{nr}^{h^+(\bar{x})} & \text{if } \mathsf{nr} > 1 \\ 1 & \text{en otro caso} \end{cases} \tag{3.2}$$

En el caso recursivo es exactamente $\frac{1}{\mathsf{nr}-1}\mathsf{nr}^{h^+(\bar{x})} - 1$ pero

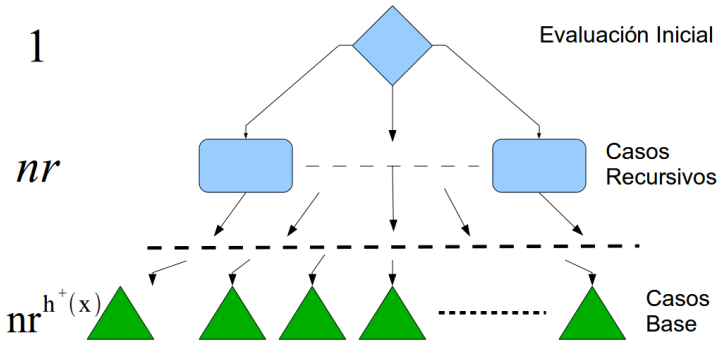

Figura 3.3: Dibujo de una evaluación pésima

Antes de seguir describiendo las técnicas para calcular las piezas individuales, vamos a ver un ejemplo de cómo aplicar la fórmula general.

Ejemplo 63 (Acumulación de Subevaluaciones)**.** Vamos a calcular la cota superior de la CR C en la Fig. 1.9. Aplicando el principio de composicionalidad, modificamos las Ecuaciones (3) y (4) sustituyendo las llamadas a D y E por estas cotas superiores D^+ y E^+:

	h^+	$costnr^+$	$costr^+$	Cota Superior Bound
$D(a_0, la_0, i_0)$	$\mathsf{nat}(la_0 - i_0)$	8	10	$8 + 10 * \mathsf{nat}(la_0 - i_0)$
$E(la_0, j_0)$	$\mathsf{nat}(la_0 - j_0 - 1)$	5	15	$5 + 15 * \mathsf{nat}(la_0 - j_0 - 1)$

Así obtenemos la variación de C mostrada en la Figura 3.1. Esta ecuación da la cota superior C^+

$$C^+(l_0, a_0, la_0, b_0, lb_0) = 2 + \mathsf{nat}(l_0) * \text{máx}(\{\mathtt{mexp}_3, \mathtt{mexp}_4\})$$

que se obtiene al aplicar la fórmula (3.1) con la h^+ calculada en el Ejemplo 72 y los expresiones $costr^+$ y $costnr^+$ del Ejemplo 78. □

3.3. Evaluación Parcial

El algoritmo de la Figura 3.2 solo es aplicable a un *CRS* directamente recursivo pero los analizadores de coste raramente generan un *CRS* así. Por eso queremos obtener, para un *CRS* $\mathcal{S}$, otro *CRS* $\mathcal{S}'$ que sea equivalente a $\mathcal{S}$ y esté en forma directamente recursiva. Para ello empleamos un algoritmo de **Evaluación Parcial** *(Partial Evaluation, PE)* [35], que transforma $\mathcal{S}$ en $\mathcal{S}'$ desplegando y suprimiendo algunas CR de $\mathcal{S}$.

Ejemplo 64 (Evaluación Parcial para *CRS* de Lista.delete)**.** COSTA [7, 8], al analizar el bucle "for" del método Vector.elimina (Figuras1.8) no genera las Ecuaciones (8) y (9) de E (Figura 1.9), sino las de la Figura 3.4:

$$
\begin{array}{lllll}
(8') & \langle E(la,j) & = & 5+F(la,j,j,la-1) & ,\{j' \geq 0\} \rangle \\
(9') & \langle F(la,j,j',la') & = & H(j',la') & ,\{j' \geq la'\} \rangle \\
(10) & \langle F(la,j,j',la') & = & G(la,j,j',la') & ,\{j' < la'\} \rangle \\
(11) & \langle H(j',la') & = & 0 & \rangle \\
(12) & \langle G(la,j,j,la-1) & = & 10+E(la,j+1) & ,\{j < la-1, j \geq 0\} \rangle
\end{array}
$$

Figura 3.4: Ecuaciones de Lista.delete antes de la Evaluación Parcial

E modela el coste de evaluar la guarda j < size del bucle for. Cada ecuación de F modela una rama de flujo: la Ecuación (9') trata el caso en que no se cumple la guarda del bucle y se sale (llamada a H); si se cumple la guarda se sigue la Ecuación (10) que llama a G. G modela el coste de la instrucción j++ y de reiterar el bucle (llamada a E). Si a partir de este *CRS* desplegamos las relaciones F, G y H obtenemos el sistema de la Fig. 1.9. □

3.3.1. Binding Time Classification

El primer paso de la PE es dividir $rel(\mathcal{S})$ en dos conjuntos $rel(\mathcal{S}) = U \cup R$, siendo U las CR que se despliegan (desplegables) y R las que se conservan (residuales). Esta división, conocida como *Binding Time Classification (BTC)*, sirve para guiar el proceso de PE y debe garantizar que la PE termina y que su resultado es el *CRS* $\mathcal{S}'$ deseado.

Para realizar la BTC se estudian las dependencias y recursividades en $\mathcal{S}$ usando el grafo de llamadas de $\mathcal{S}$ (Def. 45). Empecemos por recordar terminología de grafos: sea $G = \langle N, A \rangle$ un grafo dirigido con bucles.

- Para un subconjunto de vértices $S \subseteq N$, el subgrafo de G con respecto a S, denotado como $G|_S$, se define $G|_S = \langle S, A \cap (S \times S) \rangle$.

- Un conjunto de nodos $S \subseteq N$ es **fuertemente conexo** *(strongly connected)* si $\forall n, n' \in S$, tenemos que n' es alcanzable desde n. Una SCC S es cíclica si existe algún ciclo entre sus nodos, lo que sucede si S tiene varios elementos o si S tiene un vértice y existe un bucle en ese vértice. En el grafo de llamadas de un *CRS*, una SCC con varios vértices representa un conjunto de varias CR interdependientes e indirectamente recursivas, una SCC unitaria acíclica corresponde una CR no recursiva y una SCC unitaria cíclica a una CR directamente recursiva.

- Las **componentes fuertemente conexas** *(strongly connected components)* de G, escrito $SCC(G)$, es una partición de N en sus máximos subconjuntos fuertemente conexos.

- Sea S una componente fuertemente conexa del grafo G. Un nodo $n \in S$ es un **punto de cobertura** *(covering point CP)* del subgrafo $G|_S$ si $G|_{S\setminus\{n\}}$ es un grafo acíclico, esto es si todo ciclo en $G|_S$ pasa por n. En un grafo de llamadas, un CP de un SCC corresponde a una CR por la que pasan todos los ciclos de recursividad del SCC.

El principal obstáculo que hay en la PE es la recursividad del *CRS*. Para superarlo la BTC debe conservar algunas CR que, si se suprimieran, desaparecería la recursividad del *CRS*. En el grafo de llamadas, esto consiste en encontrar un conjunto de vértices que necesitamos suprimir de un grafo cíclico para convertirlo en uno acíclico, lo que se conoce como el problema del *feedback vertex set*. Los CP proporcionan la solución a este problema: si se suprimen del *CRS* las CR que corresponde a un CP de un SCC del grafo, entonces desaparece la recursividad del SCC.

La transformación que proponemos suprime las relaciones intermedias del *CRS* y logra la recursión directa si al menos permanece una relación de cada SCC del grafo de llamadas.

El concepto de nodos de cobertura nos sirve para definir una condición suficiente bajo la cual es posible transformar el *CRS* $\mathcal{S}$ a forma directamente recursiva. En concreto, basta con que el grafo de llamadas sea de mínima cobertura.

Definición 47 (Grafo de Mínima Cobertura)**.** Un grafo $G = \langle N, A\rangle$ es de **mínima cobertura** *(minimal coverage)* si, y solo si; todas sus SCCs tienen al menos un punto de cobertura. Un *CRS* es de mínima cobertura si su grafo de llamadas es de mínima cobertura. □

Ejemplo 65 (Grafo de mínima cobertura)**.** El grafo de llamadas $\mathcal{G}(\mathcal{S})$ del *CRS* del Ejemplo 64, se muestra en la Fig. 3.5(izquierda). Podemos ver que

$$SCC(\mathcal{G}(\mathcal{S})) = \{\{E, F, G\}, \{H\}\}$$

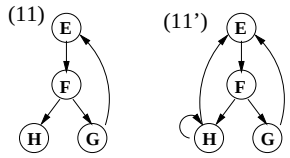

Figura 3.5: Grafo de Llamadas de Mínima Cobertura (izquierda) y G.L que no es de mínima Cobertura (derecha).

La única SCC con más de un vértice es $\{E, F, G\}$ en la que solo hay un ciclo y por tanto todos sus vértices son nodos de cobertura. □

Ejemplo 66 (Grafo que no es de mínima cobertura)**.** En el Ejemplo 64, si reemplazamos la Ecuación (11) con la (11'):

$$(11')\langle H(j', la') = 1 + H(j'', la') + E(j'', la'), \{j'' = j' - 1\}\rangle$$

obtenemos el grafo de llamadas que está en el lado derecho de la Figura 3.5. En éste, $SCC(\mathcal{G}(\mathcal{S})) = \{\{E, F, G, H\}\}$, todos los nodos son del mismo SCC y tenemos tres ciclos ($\langle E, F, G\rangle$, $\langle E, F, H\rangle$,y $\langle H\rangle$) en él, pero ningún nodo pertenece simultáneamente a los tres ciclos. □

El contraejemplo anterior muestra que algunos grafos no son de mínima cobertura, lo que significa que nuestro algoritmo no trata algunos *CRS*. Esta cuestión se discute en detalle en la Sección 3.7.

Ahora podemos definir la noción de BTC directamente recursivo, que garantiza la terminación y efectividad de la evaluación parcial.

Definición 48 (BTC Directamente Recursivo)**.** Dado un *CRS* $\mathcal{S}$ con grafo G, una BTC btc de $\mathcal{S}$ es *directamente recursiva* si para todo $S \in SCC(G)$ se cumplen las siguientes condiciones:

> **(DR)** Si $s_1, s_2 \in S$ y $s_1, s_2 \in$ btc, entonces $s_1 = s_2$. Es decir, la BTC contiene a lo sumo una CR de S. Esto garantiza que todas las recursiones en $\mathcal{S}'$ son directas.
>
> **(TR)** Si S tiene algún ciclo, esto es si contiene varias CR o una CR directamente recursiva, entonces btc contiene exactamente una $s \in S$. Esto garantiza que el desplegado termina. □

Ejemplo 67 (BTC directamente recursiva en Lista.delete)**.** Una BTC directamente recursiva para el Ej. 64 es btc $= \{E\}$. □

Por último, señalamos que en nuestro algoritmo el BTC solo señala como residuales los nodos de cobertura de las SCC, por eso las CR intermedias que no forman parte de un SCC también son desplegadas.

Ejemplo 68 (Desplegado de CR independientes)**.** En el Ejemplo 64 la BTC calculada es $\mathsf{btc} = \{E\}$, que no incluye la CR independiente H. Por ese motivo H también es desplegada y el *CRS* de la Figura 1.9 no incluye sus ecuaciones. Fíjese, sin embargo, que la CR *Del* no se ha desplegado porque ésta tiene significado externo. □

3.3.2. Desplegado

En programación funcional, la noción de **desplegado** *(unfolding)* consiste, intuitivamente, en reemplazar una llamada a una función f por su definición. En una CR esa noción no solamente debe combinar (sumar) las expresiones aritméticas sino que también debe observar la satisfactibilidad de las restricciones de tamaño.

Definición 49 (Desplegado de CR)**.** Sean $\mathcal{S}$ un *CRS*, $C \in rel(\mathcal{S})$ una CR, $C(\bar{x}_0)$ una llamada a esa CR y $\varphi_{\bar{x}_0}$ una restricción de tamaño de $\bar{x}_0$. Una especialización *(specialization)* $\langle E, \varphi\rangle$ se obtiene al desplegar $C(\bar{x}_0)$ y $\varphi_{\bar{x}_0}$ en $\mathcal{S}$, escrito $\mathsf{Unfold}(\langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle E, \varphi\rangle$, si se cumple una de estas condiciones:

(res) $(C \in \mathsf{btc} \wedge \varphi \neq \mathsf{true}) \wedge \langle E, \varphi\rangle = \langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle$.

(unf) $(C \notin \mathsf{btc} \vee \varphi = \mathsf{true}) \wedge \langle E, \varphi\rangle = \langle (e + e_1 + \ldots + e_k), \varphi' \bigwedge_{i=1..k} \varphi_i\rangle$,

donde tenemos que:

1. $\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi_C\rangle$ es una ecuación renombrada en $\mathcal{S}$ tal que φ' es satisfactible en $\mathbb{Z}$, siendo $\varphi' = \varphi_{\bar{x}_0} \wedge \varphi_C[\bar{x}/\bar{x}_0]$.
2. $\mathsf{Unfold}(\langle D_i(\bar{y}_i), \varphi'\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle e_i, \varphi_i\rangle$ para cualquier $i \in \{1, \ldots, k\}$. □

El primer caso, (**res**), es necesario para terminación. Cuando llamamos a una CR residual C, solo devolvemos la llamada inicial $C(\bar{x}_0)$ y las restricciones $\varphi_{\bar{x}_0}$, siempre que $\varphi_{\bar{x}_0}$ no sea la inicial (true). La siguiente condición se añade para imponer el paso de desplegado inicial para relaciones residuales. En todas las llamadas siguientes a Unfold las restricciones son distintas de true.

El caso (**unf**) corresponde a continuar el desplegado. El paso 1 es indeterminista pues las CRs se definen con varias ecuaciones. Mas aun, como las

expresiones están desplegadas transitivamente, el paso 2 puede dar muchas soluciones, y así el desplegado puede dar muchas respuestas. Además, si φ es insatisfactible entonces $\langle E, \varphi\rangle$ no es un desplegado válido.

Ejemplo 69 (Desplegado paso a paso de una llamada)**.** Dada la llamada inicial $\langle E(la, j), true\rangle$, obtenemos un desplegado con estos pasos:

$$\begin{array}{ll}
\langle E(la, j), true\rangle & \overset{unf}{=}_{(8')} \\
\langle 5 + F(la, j, j', la'), \{j' = j, la' = la - 1, j' \geq 0\}\rangle & \overset{unf}{=}_{(10)} \\
\langle 5 + G(la, j, j', la'), \{j' = j, la' = la - 1, j' \geq 0, j' < la'\}\rangle & \overset{unf}{=}_{(12)} \\
\langle 15 + E(la, j''), \{j < la - 1, j \geq 0\}\rangle &
\end{array}$$

La última llamada no se despliega porque $E \in \mathsf{btc}$ y $\varphi \neq true$. □

En esa definición, para cada resultado del desplegado podemos construir una ecuación *residual*. Dado $\mathsf{Unfold}(\langle C(\bar{x}_0),\ \varphi_{\bar{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle E, \varphi\rangle$, su ecuación residual correspondiente es $\langle C(\bar{x}_0) = E, \varphi\rangle$.

Usamos $\mathsf{Residuals}(\langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle, \mathcal{S}, \mathsf{btc})$ para indicar el conjunto de ecuaciones residuales de $\langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle$ en $\mathcal{S}$ con respecto a φ.

3.3.3. Evaluación parcial

Ahora, obtenemos una evaluación parcial de C recolectando todas las ecuaciones residuales de $\langle C(\bar{x}_0), \mathsf{true}\rangle$.

Definición 50 (Evaluación Parcial)**.** Dado un *CRS* $\mathcal{S}$, una CR C y una BTC btc de $\mathcal{S}$, el *CRS* $\mathcal{S}'$ definido como

$$\mathcal{S}' = \bigcup_{D \in \mathsf{btc} \cup \{C\}} \mathsf{Residuals}(\langle D(\bar{x}_0), \mathsf{true}\rangle, \mathcal{S}, \mathsf{btc})$$

es la *evaluación parcial* para C en $\mathcal{S}$ con respeto a btc. □

Esa definición proporciona un algoritmo de evaluación parcial para *CRSs*. En términos de PE [35], el algoritmo es una PE *off-line* monovoriante en el nivel global (pues para todas las relaciones residuales, la restricción inicial es true) y en el control local despliega las relaciones desplegables y solo residualiza (conserva) el resto. Notese que además de las relaciones en btc, también se generan ecuaciones adicionales para C.

Ejemplo 70 (Evaluación Parcial de $\mathsf{Lista.delete}$)**.** La evaluación parcial de las Ecuaciones del Ejemplo 64 con respecto a la llamada del Ejemplo 69 son las Ecuaciones (8,9) de la Figura 1.9. La Ecuación (9) se obtiene por los pasos de desplegado mostrados en el Ejemplo 69 y la Ecuación (8) de una derivación de desplegado donde las ecuaciones elegidas son (8',9',11). Como se esperaba, el resultado es un *CRS* directamente recursivo. □

Ahora que hemos definido el algoritmo de evaluación parcial y mostrado un ejemplo de su aplicación, demostramos que, como esperábamos, $\mathcal{S}'$ está en forma directamente recursiva (Lema 1) y es equivalente a $\mathcal{S}$ (Lema 2).

Lema 1 (Recursión directa de la Evaluación Parcial)**.** Sean $\mathcal{S}, C, \mathsf{btc}$ como en la Definición 50. La Evaluación Parcial de C en $\mathcal{S}$ con respecto a btc

1. termina en tiempo finito y
2. su resultado $\mathcal{S}'$ es directamente recursivo. □

Demostración. Primero probamos la afirmación (2). Supongamos que $\mathcal{S}'$ no es directamente recursivo, es decir que hay dos CR distintas, $D, E \in \mathit{rel}(\mathcal{S}')$ mutuamente recursivas y dos ecuaciones

$$\langle D(\bar{x}) = e + E(\bar{y}), \varphi_D \rangle \qquad\qquad \langle E(\bar{x}) = e + D(\bar{y}), \varphi_E \rangle$$

Puesto que D, E no fueron desplegadas, debe ocurrir que $D, E \in \mathsf{btc}$. Pero como D está en el mismo SCC de E, la condición (**DR**) de la Definición 48 exige que $C = D$, lo que contradice la suposición inicial.

Ahora vamos a demostrar (1) por contradicción: si la evaluación parcial no termina es porque Unfold no termina, lo que significa que hay una secuencia infinita de pasos de desplegado

$$\langle E_1, \varphi_1 \rangle \overset{unf}{=} \langle E_2, \varphi_2 \rangle \overset{unf}{=} \ldots \overset{unf}{=} \langle E_n, \varphi_n \rangle \overset{unf}{=} \langle E_{n+1}, \varphi_{n+1} \rangle \overset{unf}{=} \ldots$$

Dado que $\mathit{rel}(\mathcal{S})$ es finito y la secuencia no, debe haber alguna CR que aparezca en repetida en dos pasos E_i y E_j (siendo $j \geq i$), pero esto no es posible porque bajo la condición (**TR**) de la Def. 48 debe haber un E_k entre E_i y E_j tal que $E_k \in \mathsf{btc}$. □

Lema 2 (Corrección de la Evaluación Parcial)**.** Sean $\mathcal{S}, C, \mathsf{btc}, \mathcal{S}'$ como en la Definición 50. Entonces,

$$\forall C \in \mathsf{btc}, \forall \bar{v} \in \mathbb{Z}^n, \forall r \in \mathbb{R}^+ : \mathcal{S} \models r \in C(\bar{v}) \iff \mathcal{S}' \models r \in C(\bar{v})$$

□

Demostración. En programación lógica existen resultados de corrección de evaluación parcial como [41, 36, 35, 40, 39]. Nos basamos en esos resultados probando que la Def. 50 cumple estos requisitos:

1. *Solidez:* $\mathcal{S} \models r \in C(\bar{v}) \impliedby \mathcal{S}' \models r \in C(\bar{v})$

   Esto requiere probar la corrección del operador Unfold de la Definición 49, lo que resulta trivial pues Unfold solo reemplaza en la regla **(unf)** una llamada a una CR por su definición y propaga las restricciones. En términos de evaluación, sería *resumir* varios pasos de evaluación en $\mathcal{S}$ al mezclar una evaluación con sus derivadas.

2. *Completitud:* $\mathcal{S} \models r \in C(\bar{v}) \Longleftarrow \mathcal{S}' \models r \in C(\bar{v})$

   En las CR que permanecen, las respuestas de $\mathcal{S}$ también las da $\mathcal{S}'$. Se asegura si el conjunto de términos parcialmente evaluados cumple la condición de cierre *(closedness)* [41]. Ésta asegura que todas las posibles llamadas durante la ejecución de un *CRS* encontrarán una relación que se ajuste. Nosotros debemos asegurar que el btc garantiza esta condición.

   El cierre de los términos que son parcialmente evaluados, esto es los elementos en la btc, se deriva del hecho de que solo los términos en la btc permanecen en la CR y el resto son desplegados. Esto asegura de manera trivial que todas las posibles llamadas durante una ejecución serán cubiertas por la btc. Por último, el Lema 1 asegura que la evaluación parcial termina.

□

3.4. Acotar el número de iteraciones

El objetivo de esta Sección es definir, para una CR directamente recursiva, una expresión de coste h^+ que sea una cota superior del número de iteraciones de C. Para ello vamos a calcular una **función de rango** *(ranking function)* mediante el análisis de los bucles de C.

3.4.1. Funciones de Rango

El análisis de terminación intenta averiguar si un bucle termina. Un concepto muy importante en este estudio es el de **orden parcial fundado** *(well-founded order, WFO)*, que es aquel en que no existen secuencias infinitas estrictamente decrecientes. Uno de los resultados más importantes del estudio de terminación establece que un bucle termina si y solo si existe un orden bien fundado entre los estados del bucle tal que toda iteración del bucle implica un descenso en ese WFO.

Una forma sencilla de definir un WFO entre los estados de un bucle es emplear un orden bien fundado conocido (como $\mathbb{N}$) y proyectar en él los estados del bucle. Dicha proyección consiste en una función f que asigne a cada estado del bucle un valor en ese WFO y cumpla la condición de reducción. Dicha función recibe el nombre de **función de rango** *(ranking function, RF)*.

Para probar la terminación basta demostrar que el bucle admite una RF. Pero nosotros también necesitamos una RF f especial que sirva para calcular

h^+. En concreto, debe ser una aplicación lineal de los parámetros del bucle en el WFO $([a, +\infty), \leq_\epsilon)$ donde $a \in \mathbb{R}$ y $\epsilon \in [1, +\infty))$ y el orden $\leq_\epsilon$ se define como $r \leq_\epsilon s \leftrightarrow r = s \vee r + \epsilon \leq s$. Existe un algoritmo completo, desarrollado por Podelsky [48], para encontrar una RF lineal de $\langle C(\bar{x}) \to C(\bar{y}), \varphi \rangle$ a partir del análisis de φ.

Definición 51 (Función de Rango de una CR)**.** Sea C una CR directamente recursiva del *CRS* $\mathcal{S}$. $f_C : \mathbb{Z}^n \mapsto \mathbb{Z}$ es una función de rango para C si se cumple

$$\forall \langle C(\bar{x}) \to C(\bar{y}), \varphi \rangle \in \mathcal{S}.\mathit{Loops}(C) : \varphi \models f(\bar{x}) > 0 \wedge f(\bar{x}) \geq f(\bar{y}) + 1$$

siendo $\mathcal{S}.\mathit{Loops}(C)$ el de la Definición 39 La primera condición dice que f define un WFO entre los estados del bucle, y la segunda que f descienda en cada iteración. □

Aquí es donde importa que el análisis de tamaños conserve información importante en φ sobre el comportamiento del bucle, como la alteración de las variables en cada iteración y las condiciones en que la guarda no es aplicable.

Ejemplo 71 (Función de Rango de un bucle)**.** En la CR de la Figura 1.12, podemos inferir los siguientes bucles:

$$\begin{array}{llll} (2)\langle C_m(i,n) & \to & C_m(i+1,n) & ,\{i < n\} \quad \rangle \\ (3)\langle C_m(i,n) & \to & C_m(i,n-1) & ,\{i < n\} \quad \rangle \end{array}$$

podemos ver que en (2) el primer argumento i de C_m se incrementa en 1 y en (3) el segundo argumento n se decrementa en 1. Sea la función $f(a,b) = b - a$, podemos observar que

$$\varphi_2 \vee \varphi_3 \models f(i,n) > f(i',n') \wedge f(i,n) \geq 0$$

luego el valor de f decrece en cada iteración pero nunca por debajo de 0, luego el bucle da a lo sumo $\mathsf{nat}(f(i_0,n_0)) = \mathsf{nat}(n_0 - i_0)$ iteraciones. □

Ejemplo 72 (Función de Rango de un bucle)**.** La función $f_C(l,a,la,b,lb) = l$ es una función de rango para la CR C de la Figura 1.9. Nótese que φ'_1 y φ'_2 en los bucles de C del Ejemplo 53 contienen las restricciones $\{l > l', l > 0\}$ que bastan para garantizar que f_C es decreciente y bien fundada. □

A veces una función de rango involucra varios argumentos de la CR.

Ejemplo 73 (Función de rango de varios argumentos)**.** Considere el bucle $\langle D(a,la,i) \to D(a,la,i'), \{i' = i+1, i < la, a \geq la, i \geq 0\}\rangle$ inducido por la Ecuación (7) de la Fig. 1.9. La función $f_D(a,la,i) = la - i$ es una RF de este bucle, y cualquier otra RF de D debe incluir tanto la como i. □

3.4.2. Cotas superiores del número de iteraciones

Ahora mostramos cómo obtener $h^+(\bar{x})$ a partir de una función de rango $f_C(\bar{x})$ de C. Esto se justifica porque (1) la RF se reduce en al menos una unidad en cada iteración cuando se aplica en dos llamadas consecutivas (rango $\mathbb{Z}$) y (2) siempre es mayor que 0 en los casos recursivos.

El primer obstáculo es que h^+ solo admite rango natural mientras que f_C también puede devolver valores enteros negativos. Esto sucede con argumentos $\bar{v}$ que no satisfacen las guardas de los bucles (casos base o fuera de dominio). Para resolverlo, emplearemos el operador nat, que garantiza un resultado en $\mathbb{N}$.

Lema 3 (Cota Superior del número de iteraciones basada en Función de Rango)**.** Sea $f_C(\bar{x})$ una función de rango para la CR C. Entonces, para cualquier evaluación $C(\bar{v})$, $\mathsf{nat}(f_C(\bar{v})) \geq h(C(\bar{v}))$. □

Demostración. Vamos a proceder por inducción sobre el valor de $h(C(\bar{v}))$.

- Caso Base $h = 0$. Resulta trivial que $\mathsf{nat}(f_C(\bar{v})) \geq 0$ pues el operador nat solo devuelve valores no negativos.
- Caso Inductivo: $h > 0$. Vamos a proceder por contradicción. Supongamos que existe una evaluación de $C(\bar{v})$ tal que $h(C(\bar{v})) = n > \mathsf{nat}(f_C(\bar{v}))$. Esta evaluación tendrá una rama con $n + 1$ iteraciones $C(\bar{x}_0), \ldots, C(\bar{x}_n)$, siendo $\bar{x}_0 = \bar{v}$. A partir de la definición de RF, para todo $i < n$ se tiene que $f_C(\bar{v}_i) - f_C(\bar{v}_{i+1}) \geq 1$ y $f_C(\bar{v}_i) > 0$. Luego $f_C(\bar{v}) > n = h(T)$, lo que contradice la hipótesis $f_C(\bar{v}) < h(T)$.

□

Recursión Lineal

Sea f_C una función de rango para la CR C que satisface la condición de reducción aritmética con razón $r = 1$. Entonces podemos usar esta primera fórmula de la cota superior

$$h^+(\overline{x}) = \mathsf{nat}(f_C(\overline{x}))$$

Aunque esta fórmula proporciona una cota superior correcta, a veces podemos refinarla y obtener otra cota más ajustada. Por ejemplo, si la reducción aritmética de la función de rango siempre supera una constante $\delta > 1$ entonces una fórmula de cota superior más ajustada es

$$h^+(\overline{x}) = \delta^{-1} * \mathsf{nat}(f_C(\overline{x}))$$

Recursión Logarítmica

Un caso muy interesante sucede cuando cada bucle $\langle C(\bar{x}) \to C(\bar{y}), \varphi \rangle \in \mathit{Loops}(C)$ satisface $\varphi \models f_C(\bar{x}) \geq k * f_C(\bar{y})$ siendo $k > 1$ una constante real. Es decir, cuando la función de rango presenta una propiedad de reducción geométrica. En este caso, el número de iteraciones está acotado por

$$h^+(\overline{x}) = \log_k(\mathsf{nat}(f_C(\bar{v})) + 1)$$

Al número k se conoce como **factor logarítmico** de C, y decimos que C tiene una recursión k-logarítmica. En caso contrario, decimos que tiene recursión lineal.

Ejemplo 74 (Recursión Logarítmica)**.** Si modificamos el *CRS* de la Figura 1.12, cambiando φ_2 y φ_3 por $\varphi_2' = \{i{<}n, i'{=}i{*}2\}$ y $\varphi_3' = \{i{<}n, n'{=}n/2\}$ entonces $\mathsf{l_r} \leq \log_k(\mathsf{nat}(n{-}i){+}1)$. □

3.5. Acotando los Costes Locales

El objetivo de esta sección es calcular $\mathit{costnr}^+(\bar{x})$ y $\mathit{costr}^+(\bar{x})$. Para ello vamos a maximizar las expresiones de coste locales de las ecuaciones de la CR C. Pero el valor de esas expresiones depende del valor de las variables locales en cada subevaluación de $C(\bar{v})$, y es muy complicado determinar esos valores en cada iteración. Es más eficaz[10] calcular un invariante de los bucles de C y maximizar los costes locales en función de $\bar{v}$.

3.5.1. Invariantes

Nuestro primer paso es calcular un invariante de la CR C. En la Subsección 2.5.2 se definió qué es un sistema de bucles y qué es un invariante (Definición 35) y se dio la fórmula del invariante exacto I_C (Definición 36). Pero esta fórmula casi nunca es aplicable porque consiste en una operación de infinitos operandos (todas las potencias del bucle).

Ejemplo 75. Calculamos el invariante exacto I_C de los bucles del Ej. 53. Tras tres iteraciones I_C incluye $\{I_0, \ldots, I_6\}$. Subsiguientes iteraciones añaden nuevos contextos en los que la o lb seguirán disminuyendo. Por tanto, I_C tiene infinitos elementos y la definición no es aplicable. □

Necesitamos otro algoritmo para calcular un invariante del bucle que sea eficiente y terminante aunque obtenga un invariante más débil. Para ello

usamos la interpretación abstracta (Cousot [24]) en el dominio de los poliedros convexos y vamos a aproximar I_C en un solo bucle (poliedro convexo) $I_C^\alpha = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi' \rangle$ que también sea un invariante del bucle.

Para ello vamos a modificar la fórmula del invariante exacto de la Definición 36 reemplazando el operador $\cup$, unión exacta, por la composición de dos operadores: la **unión convexa** *(convex-hull)* para reducir todos los contextos de una iteración a un solo contexto convexo, y el operador de **ensanchamiento** *(widening)* [24] permite abstraer y generalizar los resultados, y garantiza la terminación y eficiencia del cálculo.

Ejemplo 76. En el Ejemplo 75, aproximamos I_C con:

$$I_C^\alpha = \{\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l \leq l_0, a = a_0, la \leq la_0, b = b_0, lb \leq lb_0\}\rangle\}$$

□

3.5.2. Cotas superiores de Expresiones de Coste

Puesto que los costes locales de cada aplicación son instancias de los costes locales de las ecuaciones de coste, procedemos por acotar esos costes locales en las ecuaciones. Más arriba, al presentar las expresiones de coste mencionamos, en la Proposición 2, que éstas son monótonas sobre sus subexpresiones nat. Esto facilita el cálculo de cotas superiores para expresiones de coste: para encontrar una cota superior de e en función de unas variables $\bar{x}_0$, basta con acotar cada expresión lineal $l \in lins(e)$ en función de esas variables bajo las condiciones del invariante. Esto nos lleva al problema de acotar una expresión lineal en función de unas variables y bajo un sistema de restricciones lineales. Éste ha sido estudiado en programación lineal, y se conoce desde hace tiempo algoritmos correctos y completos. La inferencia puede calcularse algorítmicamente con herramientas tales como [12]).

Ejemplo 77. Queremos calcular una cota superior de la expresión $\mathsf{nat}(la - j - 1)$, que aparece en la Ecuación (3) de la Figura 3.1, en función de los parámetros de la llamada inicial $\langle l_0, a_0, la_0, b_0, lb_0 \rangle$. Hemos inferido que

$$\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l \leq l_0, a = a_0, \underline{la \leq la_0}, b = b_0, lb \leq lb_0\}\rangle$$

es un invariante de C, y vemos que el valor máximo que la puede tomar es la_0. Asimismo, la guarda φ de la Ecuación (3) nos dice que $j \geq 0$. Puesto que el valor máximo de $la - j - 1$ se da cuando la es máximo y j mínimo, se deduce que $la_0 - 1$ es una cota superior de $la - j - 1$. □

1: **function** *ub_exp*(e,$\bar{x}_0$,φ,ψ)
2: $\texttt{mexp} = e$
3: **for all** $\mathsf{nat}(f) \in e$ **do**
4: $\phi = \bar{\exists}\bar{x}_0, y.(\varphi \wedge \psi \wedge y = f)$ // y es una variable nueva
5: **if** $\exists f'$ tal que $vars(f') \subseteq \bar{x}_0$ y $\phi \models y \leq f'$ **then** $\texttt{mexp} = \texttt{mexp}[\mathsf{nat}(f)/\mathsf{nat}(f')]$
6: **else return** ∞
7: **return** mexp

Figura 3.6: Algoritmo de Maximización de Expresiones de Coste

Dada una ecuación de coste

$$\langle C(\bar{x}) = e + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle$$

y un invariante correcto $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle$, el algoritmo de la Figura 3.6 computa una cota superior de e maximizando sus componentes nat: La función calcula una cota superior f' para cada expresión lineal f que aparece dentro de una función nat y después reemplaza en e todas esas expresiones f por sus respectivas cotas superiores (línea 5). Si no puede encontrar una cota superior, el método devuelve ∞ (linea 6).

Ejemplo 78. Al aplicar *ub_exp* a las expresiones de coste e_3 y e_4, que aparecen en las Ecuaciones (2) y (3) de la Fig. 3.1, con respecto al invariante computado en la Sección 3.5.1, puede hacerse maximizando sus subexpresiones nat. Del mismo modo que antes hemos encontrado una cota de $la - j - 1$, también encontramos para $lb - j - 1$, la, lb las cotas superiores $lb_0 - 1$, la_0 y lb_0, respectivamente. Y por tanto,

$$\begin{array}{rcl} e_3 & \leq & \texttt{mexp}_3 = 38 + 15 * \mathsf{nat}(la_0 - 1) + 10 * \mathsf{nat}(la_0) \\ e_4 & \leq & \texttt{mexp}_4 = 37 + 15 * \mathsf{nat}(lb_0 - 1) + 10 * \mathsf{nat}(lb_0) \end{array}$$

□

Ahora podemos usar un invariante para maximizar los costes locales de cada iteración en función de los argumentos de la llamada inicial.

$$worst(\{rec_1, \ldots, rec_{nr}\}) = \text{máx}(maximize(\mathcal{I}, \{rec_1, \ldots, rec_{nr}\}))$$

El operador *maximize* toma una restricción de tamaño φ y una serie de expresiones a maximizar y calcula el valor máximo de cada expresión en función de los variables iniciales de cada invariante.

Ejemplo 79. En la CR de la Figura 1.12 calculamos

$$worst(\{rec_1, rec_2\}) = \text{máx}(maximize(\mathcal{I}, \{\mathsf{nat}(n-i), \mathsf{nat}(i)\}))$$

cuyo resultado es $\text{máx}(\{\mathsf{nat}(n_0-i_0), \mathsf{nat}(n_0-1)\})$. El coste local del caso base es constante y no requiere maximización. □

Propiedad 8. Sea la CR $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ donde $\mathcal{S}_1$ y $\mathcal{S}_2$ son respectivamente las ecuaciones básicas y recursivas de C. Si conocemos un invariante de $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle$ y definimos las expresiones

$$\begin{array}{rl} costnr^+(\bar{x}_0) = \text{máx}(\{ub\_exp(e, \bar{x}_0, \varphi, \psi) \mid \langle C(\bar{x}) = e & , \varphi\rangle \in \mathcal{S}_1\}) \\ costr^+(\bar{x}_0) = \text{máx}(\{ub\_exp(e, \bar{x}_0, \varphi, \psi) \mid \langle C(\bar{x}) = e \quad + \sum_{j=1}^{k} C(\bar{y}_j) & , \varphi\rangle \in \mathcal{S}_2\}) \end{array}$$

Entonces, para cualquier evaluación de la llamada $C(\bar{v})$ se cumple que

- en toda subevaluación recursivas de coste local r, $costr^+(\bar{v}) \geq r$ y
- $costnr^+(\bar{v}) \geq r$ en toda subevaluación básica de coste local r. □

3.6. Acumulación por Niveles

La fórmula de Acumulación de Subevaluaciones (Proposición 3) permite obtener cotas siempre correctas y a menudo precisas cuando se emplea con los *CRS* que se generan en el análisis de una clase amplia de programas. Pero se obtiene resultados muy imprecisos para ciertos *CRSs*, característicos de programas *divide y vencerás*.

Ejemplo 80. Sea C una CR definida con estas dos ecuaciones:

$$\begin{array}{lcl} \langle C(x) & = & 0, \{x \leq 0\}\rangle \\ \langle C(x) & = & \mathsf{nat}(x) + C(x_1) + C(x_2) \\ & , & \{x \geq x_1 + x_2 + 1, x \geq 2x_1, x \geq 2x_2, x_1 \geq 0, x_2 \geq 0\}\rangle \end{array}$$

Al aplicar la fórmula de acumulación de subevaluaciones de la Proposición 3, obtenemos la cota superior

$$C^+(x) = \mathsf{nat}(x) * (2^{\log_2(\mathsf{nat}(x)+1)}) = \mathsf{nat}(x)^2$$

que es demasiado imprecisa. □

Intuitivamente, en ellos existe recursión múltiple por lo que el número de subevaluaciones es exponencial. Pero los costes locales descienden muy rápido en cada iteración, por lo que la suma de los costes de cada nivel no se incrementa en cada iteración y, así, el coste pésimo es polinómico.

Esta sección describe la fórmula de cota superior por *acumulación por niveles*, que busca acotar simultáneamente el número de niveles y el coste total por nivel. Esta fórmula obtiene resultados más precisos para algunos de esos *CRSs* mencionados.

Proposición 4 (Cota Superior de Acumulación por Niveles)**.** Sea C una CR. La expresión de coste C^+ definida como

$$C^+(\bar{x}) = (h^+(\bar{x}) + 1) * costl^+(\bar{x})$$

es una cota superior de C si para cualquier evaluación $C(\bar{v})$ si se cumple que

- $h^+(\bar{x})$ es una cota superior del número de niveles de $C(\bar{v})$.
- Cada subnivel i de$C(\bar{v})$ satisface que $costl^+(\bar{v}) \geq \mathsf{Sum\_Level}(T, i)$. □

Demostración. Es una consecuencia inmediata de la definición de cota superior. □

Ya hemos dicho cómo calcular $h^+(\bar{x})$ en la Sección 3.4. Encontrar $costl^+$ no es tan fácil. Por eso esta fórmula no es tan aplicable como la Acumulación de Subevaluaciones.

CRS Divide y Vencerás

Ahora caracterizamos una clase de *CRS*, que llamaremos **divide y vencerás** *(divide and conquer)*, para los que podemos encontrar una expresión $costl^+$ y así aplicar la fórmula de acumulación por niveles.

Intuitivamente, en un programa de esta clase el coste total de cada nivel está acotado por el coste total del nivel superior y, por tanto, por el coste local de la aplicación raíz (que resulta de instanciar las ecuaciones recursivas).

Una condición suficiente es que el coste local de toda evaluación es una cota superior de la suma de los costes de sus evaluaciones derivadas. Para comprobarlo se debe mirar toda posible combinación de una ecuación recursiva con otras en sus aplicaciones derivadas.

Lema 4 (Condición suficiente de Divide y Vencerás)**.** Sea C una CR independiente. Si para cualquier $\langle e, e', \psi \rangle \in Child\_Exps(C)$ y cualquier $\sigma : vars(e) \cup vars(e') \mapsto \mathbb{Z}$ tal que $\sigma \models \psi$ se cumple que $[\![e]\!]_\sigma \geq [\![e']\!]_\sigma$, entonces para cualquier llamada $C(\bar{v})$, cualquier evaluación $T \in Trees(C(\bar{v}), \mathcal{S})$, y cualquier nivel k, se cumple que $\mathsf{Sum\_Level}(T, k) \geq \mathsf{Sum\_Level}(T, k+1)$. □

Demostración. Supongamos que hay una evaluación de $C(\bar{v})$, en la que para un nivel k, $\mathsf{Sum\_Level}(T,k) < \mathsf{Sum\_Level}(T,k+1)$. Esto significa que el nivel k contiene una aplicación en la cual cada subevaluación verifica $r < r_1 + \cdots + r_n$. Si esta aplicación es una instancia de $\mathcal{E} = \langle C(\bar{x}) = e + \sum_{i=1}^{m} C(\bar{y}_i), \varphi\rangle$ y $\langle C(\bar{y}_i) = e_i + \sum_{j=1}^{m_i} C(\bar{z}_j), \varphi_i\rangle \in \mathcal{S}$ se usó para asociar cada llamada $C_i(\bar{y}_i)$ en $\mathcal{E}$. Por lo tanto existe σ tal que $\sigma \models \varphi \wedge \varphi_1 \wedge \cdots \wedge \varphi_m \models \bar{x} = \bar{v} \wedge \bar{y}_1 = \bar{v}_1 \wedge \cdots \wedge \bar{y}_m = \bar{v}_m$ y $[\![e]\!]_\sigma < [\![e_1 + \cdots + e_m]\!]_\sigma$, lo que contradice la hipótesis. □

Intuitivamente, si en cada subevaluación de $C(\bar{v})$ hay una tupla $\langle e, e', \psi\rangle \in Child\_Exps(C)$ y una substitución $\sigma : vars(e) \cup vars(e') \mapsto \mathbb{Z}$ tal que $\sigma \models \psi$, $[\![e]\!]_\sigma$ es igual a su coste local, y $[\![e']\!]_\sigma$ es igual a la suma de los costes locales de sus sucesores.

Ejemplo 81. En la CR C del Ej. 80, $Child\_Exps(C)$ es :

$$Child\_Exps(C) = \left\{ \begin{array}{l} \langle \mathsf{nat}(x), 0, \varphi \wedge x_1 \leq 0 \wedge x_2 \leq 0\rangle \\ \langle \mathsf{nat}(x), \mathsf{nat}(x_1), \varphi \wedge x_1 \leq 0 \wedge \varphi_2\rangle \\ \langle \mathsf{nat}(x), \mathsf{nat}(x_2), \varphi \wedge \varphi_1 \wedge x_2 \leq 0\rangle \\ \langle \mathsf{nat}(x), \mathsf{nat}(x_1) + \mathsf{nat}(x_2), \varphi \wedge \varphi_1 \wedge \varphi_2\rangle \end{array} \right\}$$

donde φ_1, φ_2 son renombramientos de φ, salvo por x_1 (x_2). □

Ejemplo 82. Considérese otra vez el conjunto $Child\_Exps(C)$ del Ejemplo 80. Véase que en la cuarta tupla, como $\varphi \wedge \varphi_1 \wedge \varphi_2 \models \{x \geq 2 * x_1, x \geq 2 * x_2\} \models \{x \geq x_1 + x_2\}$, por lo que $\mathsf{nat}(x) \geq \mathsf{nat}(x_1) + \mathsf{nat}(x_2)$. Si hacemos una comprobación semejante en las otras tuplas, entonces vemos que C cumple la condición de divide y vencerás (Lema 4). □

Propiedad 9. Sea C una CR independiente que satisface la condición de divide y vencerás del Lema 4, E un conjunto definido como

$$E = \left\{ ub\_exp(e, \bar{x}_0, \varphi, \{\bar{x}_0 = \bar{x}\}) \mid \left\langle C(\bar{x}) = e + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \right\rangle \in \mathcal{S} \right\}$$

y $costl^+(\bar{x}) = \text{máx}(E)$. Entonces, para cualquier evaluación de $C(\bar{v})$, en el nivel k se cumple que $costl^+(\bar{v}) \geq \mathsf{Sum\_Level}(C(\bar{v}, k)$. □

Ejemplo 83. Considere de nuevo la CR C del Ej. 80. Al calcular el conjunto E del Teorema 9 se obtiene $\{\mathsf{nat}(x), 0\}$, luego $costl^+(x) = \mathsf{nat}(x)$. Usando las técnicas de la Sección 3.4 podemos calcular $l^+(x) = \lceil \log_2(\mathsf{nat}(x)+1)\rceil + 1$ y así obtener

$$C^+(x) = \mathsf{nat}(x) * (\lceil \log_2(\mathsf{nat}(x) + 1)\rceil + 1)$$

como la cota superior de C. □

3.7. Incompletitud en Análisis de Coste

Consideremos globalmente el proceso de análisis de coste presentado en la Sección 1.3. El problema de obtener una expresión de coste que sea cota superior del coste de un programa es estrictamente más difícil que la terminación.

Esto se explica por el hecho de que obtener una cota superior de un programa en el que cada ecuación recursiva tiene asociado un coste local no nulo implica la terminación del programa que la CR modela. Por tanto, el algoritmo de resolución es necesariamente incompleto y en ocasiones no podrá generar una cota superior.

Ciertamente, esto puede suceder porque el coste del programa sea realmente infinito, como el número de instrucciones que ejecuta un programa no terminante. Pero aun si el coste es finito, podemos no encontrar una cota superior debido a la imprecisión inherente al análisis de coste.

En la primera fase, de calcular el *CRS*, se emplea interpretación abstracta para aproximar cuestiones indecidibles como el *aliasing*. Para saber más sobre la incompletitud de esta fase, puede consultarse el artículo sobre COSTA de Albert et al. [7].

La fase de resolución también es incompleta: incluso si una CR admite una cota superior en forma cerrada, puede que sea imposible calcularla. Ben-Amram demuestra en [14], que un problema más simple, la terminación de un *CRS* donde las ecuaciones tienen a lo sumo una llamada y las restricciones son de la forma $x - y \leq c$, es indecidible. Una discusión detallada sobre decidibilidad de bucles simples con restricciones enteras puede encontrarse en el artículo de Braverman [19].

En nuestro método hay tres fuentes de incompletitud, que pasamos a describir a continuación.

3.7.1. Evaluación Parcial

Hemos mencionado que para aplicar el principio de composicionalidad primero debemos pasar el *CRS* a forma directamente recursiva. Esto solo podemos hacerlo si el grafo de llamadas de ese sistema es de mínima cobertura, lo que no sucede en algunos *CRS* como el que viene a continuación:

Ejemplo 84 (Incompletitud de la Evaluación Parcial)**.** Este *CRS* no tiene ningún nodo de cobertura,

$$\begin{array}{lll} \langle C(n) & = C(n-1) + D(n-1) & , \{n > 0\} \quad \rangle \\ \langle D(n) & = D(n-1) + C(n-1) & , \{n > 0\} \quad \rangle \end{array}$$

pese a que las CR C, D son terminantes. □

Es importante señalar que esta fase sí es completa para CR extraídas de bucles estructuradas y de patrones de recursión comunes a la mayoría de programas. Debe quedar claro, no obstante, que la aplicabilidad de la evaluación parcial no está relacionada con el uso de instrucciones de salto, puesto que lo que se analiza es el código binario. Por tanto, el uso de instrucciones `break` y `continue` en lenguajes como JAVA o C no supone ningún problema, pues se puede construir el grafo de control de flujo y convertirlo a forma recursiva. La incompletitud de la Evaluación Parcial solo ocurre cuando se tienen grupos de relaciones mutuamente recursivas, como las del Ejemplo.

3.7.2. Funciones de Rango

Dado que la existencia de funciones de rango es una prueba de la terminación, este cálculo será necesariamente incompleto.

En nuestro método usamos un algoritmo completo de Podelski y Ribaschenko [48] que infiere funciones de rango lineales. El mismo Podelski señaló en ese artículo que hay bucles lineales, como

$$\langle C(n) \to C(-2 * n + 10), \{n \geq 0\} \rangle$$

que no admite una función de rango lineal. Para acotar bucles como éste y otros más complejos necesitaríamos inferir funciones de rango más sofisticadas (por ejemplo cuadráticas) pero esto no mejora demasiado la capacidad.

3.7.3. Invariantes

Para maximizar los costes locales de cada iteración en este método se calcula un invariante que es una restricción lineal entre los parámetros iniciales y las variables de cada iteración. A veces ese invariante no da información suficiente para maximizar las expresiones locales, como sucede en el siguiente ejemplo:

Ejemplo 85 (Incompletitud del cálculo de Invariantes)**.** Consideremos esta CR:

$$\begin{array}{lll} \langle C(n,m) = & m & , \{n{=}0\} \quad \rangle \\ \langle C(n,m) = & C(n',m') & , \{n'{=}n{-}1, m'{=}2{*}m, n{>}0\} \quad \rangle \end{array}$$

El valor de m en el caso base será $(2^n) * m_0$. □

En principio, podríamos usar métodos para inferir invariantes polinómicos, pero para eso necesitaríamos otro algoritmo de maximización diferente.

Capítulo 4

Cotas Superiores Asintóticas

En este Capítulo trata del uso de los órdenes asintóticos en el análisis de coste. Empezamos por definir las relaciones de orden asintótico entre funciones de varias variables en la Sección 4.1. En la Sección 4.4) definimos las relaciones de orden asintótico entre expresiones de coste. La Sección 4.3 presenta un algoritmo para reducir una expresión de coste a su forma asintótica simplificada, y la Sección 4.4 muestra cómo integrar dicha simplificación en la resolución de relaciones de coste y qué resultados se obtienen.

4.1. Órdenes Asintóticos Multivariable

Empezamos por ampliar las conocidas definiciones de $\mathcal{O}$ Θ, ya mencionadas en La Sección 1.5, a funciones multivariable $\mathbb{N}^n \mapsto \mathbb{R}^+$.

Definición 52 (Orden Asintótico de Funciones)**.** Dadas $f, g : \mathbb{N}^n \mapsto \mathbb{R}^+$, decimos que $\boxed{f \in \mathcal{O}(g)}$ si y solo si existe alguna constante real $c > 0$ y un número $m \geq 1$ tales que para cualquier $\bar{v} \in \mathbb{N}^n$ tal que $v_i \geq m$, se cumple que $f(\bar{v}) \leq c * g(\bar{v})$.

Escribiremos $f \preceq g$ para abreviar $f \in \mathcal{O}(g)$. Podemos ver que $\preceq$ es una relación de orden. □

Definición 53 (Equivalencia Asintótica de Funciones)**.** Dadas $f, g : \mathbb{N}^n \mapsto \mathbb{R}^+$, decimos que $\boxed{f \in \Theta(g)}$: si y solo si existen dos constantes reales $c_1 > 0$ y $c_2 > 0$ y un número $m \geq 1$ tal que, para cualquier $\bar{v} \in \mathbb{N}^n$ cuyas componentes cumplen $v_i \geq m$, se cumple que $c_1 * g(\bar{v}) \leq f(\bar{v}) \leq c_2 * g(\bar{v})$. □

Escribiremos $f \simeq g$ para abreviar $f \in \Theta(g)$. Podemos ver que $\simeq$ es una relación de equivalencia, y por tanto define una partición de $\mathbb{N}^n \mapsto \mathbb{R}^+$ en varias clases de equivalencia u **órdenes asintóticos** *(asymptotic orders)*, cada uno con infinitos elementos.

4.2. Órdenes de Expresiones de Coste

4.2.1. Expresiones nat-free

En estas notaciones asintóticas se supone que las funciones son monótonas sobre todos sus argumentos, por cuanto la función, a menos que sea constante, tiende a ∞ cuando todas sus entradas tienden a ∞. Esta suposición no se cumple necesariamente en los CRs obtenidos tras analizar programas reales.

Ejemplo 86 (Comportamiento asintótico del nat)**.** Considérese el bucle del programa de la Figura 1.12. Claramente, el coste de ejecución del programa aumenta con el número de iteraciones del bucle, cuya cota superior es el valor de su función de rango $n - i$. Por tanto, para poder preservar el comportamiento asintótico del programa deberíamos estudiar el caso en que $\mathsf{nat}(n-i)$ tiende a ∞, como sucede si n tiende ∞ e i permanece constante, pero no si n e i tienden a ∞ manteniendo una diferencia constante. □

Como se ve en este ejemplo, debemos considerar el comportamiento asintótico tratando cada subexpresión nat de manera atómica, como una de las variables que tienden a infinito. Para ello nosotros usamos la noción de expresiones de coste nat-free, donde cada expresión nat se reemplaza con una variable de dominio natural. Así se garantiza un uso consistente de la definición original puesto que, como se pretendía, ignorando las irregularidades que f puede presentar para valores pequeños (menores a la frontera m), el valor de la función se incrementa para valores mayores de sus argumentos.

Definición 54 (Expresiones de Coste nat-free)**.** Dada una expresión de coste e, su representación nat-free es la expresión $\tilde{e}$ que se obtiene tras aplicar estos tres pasos:

1. Cada subexpresión $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n + c)$ que aparece como exponente se reemplaza con $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n)$;

2. Las demás subexpresiones $\mathsf{nat}(a_1x_1 + \cdots + a_nx_n + c)$ se reemplazan por $\mathsf{nat}(\frac{a_1}{b}x_1 + \cdots + \frac{a_n}{b}x_n)$, siendo b el *máximo común divisor* (gcd) de $|a_1|, \ldots, |a_n|$, y $|\cdot|$ el operador de valor absoluto;

3. Las expresiones nat son reemplazadas por nat-variables, representadas con letras mayúsculas latinas $A, B, \ldots$. Se debe cumplir que nat iguales se reemplazan por la misma variable y nat distintos por distintas variables.

□

Resulta importante distinguir los casos 1 y 2 y tratarlos de manera separada, porque si confundimos una subexpresión nat polinómica con una exponencial, podríamos obtener un resultado incorrecto.

Ejemplo 87 (Diferencia entre nat exponencial y polinómico)**.** Si bien se cumple que $2^{\mathsf{nat}(2x+1)} \in \mathcal{O}(2^{\mathsf{nat}(2x)})$, así como también $\mathsf{nat}(2x) \in \mathcal{O}(\mathsf{nat}(x))$, no es cierto que

$$2^{\mathsf{nat}(2x)} \notin \mathcal{O}(2^{\mathsf{nat}(x)})$$

puesto que $2^{nat(2x)} = 4^x$ y $4^x \notin \mathcal{O}(2^x)$. Por eso no se puede simplificar $2^{\mathsf{nat}(2x)}$ a $2^{\mathsf{nat}(x)}$. □

Cuando $\mathsf{nat}(a_1x_1+\cdots+a_nx_n+c)$ no es un exponente, podemos suprimir su constante c y normalizar los coeficientes a_i al dividirlos por el *mcd* de sus valores absolutos. Intuitivamente, lo que hacemos es que para expresiones nat cuyos argumentos son expresiones lineales *paralelas* (que difieren en una constante) o directamente proporcionales con razón positiva, se obtiene una forma canónica común que resulta ser la expresión más sencilla que pertenece al mismo orden exacto de complejidad.

Ejemplo 88 (Ejemplos sencillos de transformación nat-free)**.** Algunos ejemplos de lo anteriormente comentado:

- Las expresiones $\mathsf{nat}(x), \mathsf{nat}(x-1), \mathsf{nat}(2x+1)$ pueden representarse todas con la misma variable nat, y de hecho todas están en $\Theta(nat(x))$.
- Tanto $\mathsf{nat}(2x+3)$ como $\mathsf{nat}(3x+5)$ admiten la representación canónica $\mathsf{nat}(x)$
- Similarmente sucede con $\mathsf{nat}(2x+4y)$ y $\mathsf{nat}(3x+6y)$ que son proporcionales a $\mathsf{nat}(x+2y)$. □

Gracias a estas transformaciones reducimos el número de variables nat que se usan en la representación nat-free, y por tanto obtenemos expresiones más sencillas.

Ejemplo 89 (Transformación nat-free)**.** Dada la siguiente función de coste

$$5+7{*}\mathsf{nat}(3x+1){*}\,\text{máx}(\{100{*}\mathsf{nat}(x)^2{*}\mathsf{nat}(y)^4, 11{*}3^{\mathsf{nat}(y-1)}{*}\mathsf{nat}(x+5)^2\})+ 2{*}\log(\mathsf{nat}(x+2)){*}2^{\mathsf{nat}(y-3)}{*}\log(\mathsf{nat}(y+4)){*}\mathsf{nat}(2x{-}2y)$$

su representación nat-free es:

$$5{+}7*A{*}\,\text{máx}(\{100*A^2{*}B^4, 11*3^B{*}A^2\}){+}2{*}\log(A){*}2^B{*}\log(B){*}C$$

correspondiendo las variables-nat a:

$$A \mapsto \mathsf{nat}(x) \qquad B \mapsto \mathsf{nat}(y) \qquad C \mapsto \mathsf{nat}(x-y)$$

□

4.2.2. Órdenes asintóticos de Expresiones de Coste

Definición 55 (Relación Asintótica de Expresiones de Coste)**.** Dado un conjunto de expresiones de coste $E=\{e_1, e_2\}$ y su representación nat-free $\tilde{E}=\{\tilde{e}_1, \tilde{e}_2\}$, decimos que

$$e_1 \in \mathcal{O}(e_2) \leftarrow \tilde{e}_1 \in \mathcal{O}(\tilde{e}_2) \qquad\qquad e_1 \in \Theta(e_2) \leftarrow \tilde{e}_1 \in \Theta(\tilde{e}_2)$$

□

Esta definición ajusta la Definición. 53 para su uso con expresiones de coste. Sencillamente asevera que para poder decidir la relación asintótica entre dos expresiones de coste, debemos comprobar esa relación entre sus respectivas representaciones nat-free. Véase como se debe obtener las representaciones simultáneamente, para que las subsexpresiones lineales que comparten (o que son paralelas) se representen con las mismas variables-nat.

A veces, a una expresión de coste se adjunta una restricción de tamaño φ, también llamada **restricción de contexto**, que especifica una clase de valores de entrada para los que está definida esa expresión.

Ejemplo 90 (Restricción de Contexto Asintótica)**.** La expresión de coste del Ejemplo 89 puede tener como restricción de contexto $\varphi=\{x \geq y, x \geq 0, y \geq 0\}$, que especifica que solo se aceptan valores de entrada no negativos que cumplen $x \geq y$. □

En análisis de coste, esas restricciones pueden ser el resultado de un analizador o bien pueden ser aserciones dadas por el usuario. De cualquier manera, la información de la restricción de contexto puede usarse para obtener cotas asintóticas más exactas. Por ejemplo, puede usarse para asegurar que una expresión nat es una cota superior asintótica de otras.

Ejemplo 91 (Uso de la restricción de contexto)**.** Si la restricción de contexto establece que $x \geq y$, entonces cuando tanto $\mathsf{nat}(x)$ como $\mathsf{nat}(y)$ crecen hacia el infinito, sabemos que $\mathsf{nat}(x)$ subsume asintóticamente a $\mathsf{nat}(y)$. Esta información puede servir para obtener cotas asintóticas más precisas. □

En adelante, dadas dos nat-expresiones (representadas por sus correspondientes nat-variables A y B), decimos que $\varphi \models A \succeq B$ si A subsume asintóticamente a B. cuando ambas tienden hacia ∞.

4.2.3. Propiedades de $\mathcal{O}$ y Θ

Vamos a ver ahora algunas de las propiedades de los órdenes $\mathcal{O}$ y Θ. Muchas de estas ya han sido tratadas y demostradas para las funciones de una variable, y resultan tan triviales que no vamos a poner aquí su demostración.

Propiedad 10 (Orden y Equivalencia Concreta y Asintótica)**.** El orden y equivalencia asintóticas son más débiles que el orden y la equivalencia de expresiones de coste. Para cualquier par de expresiones e_1, e_2 nat-free se cumple que $e_1 \leq e_2 \Rightarrow e_1 \in \mathcal{O}(e_2)$ y $e_1 = e_2 \Rightarrow e_1 \in \Theta(e_2)$. □

Propiedad 11 (Invarianza a Constantes Multiplicativas)**.** Para cualquier número r en $\mathbb{R}^+$ se cumple que $e * k \in \Theta(e)$. □

Propiedad 12 (Propiedades Asintóticas de la Suma)**.** Sean e_1, e_2, e_3, e_4 expresiones de coste nat-free. Entonces

1. Si $e_1 \in \mathcal{O}(e_3)$ y $e_2 \in \mathcal{O}(e_4)$ entonces $e_1 + e_2 \in \mathcal{O}(e_3 + e_4)$.
2. Si $e_1 \in \Theta(e_3)$ y $e_2 \in \Theta(e_4)$ entonces $e_1 + e_2 \in \Theta(e_3 + e_4)$.
3. Si $e_1 \in \mathcal{O}(e_2)$ entonces $e_1 + e_2 \in \Theta(e_2)$.
4. Si $e > 0$ entonces $\forall r \in \mathbb{R}^+$ se cumple que $e + r \in \Theta(e)$.

□

Propiedad 13 (Equivalencia Asintótica entre Max y Suma)**.** Sea un conjunto $\{e_1, ..., e_n\}$ de expresiones de coste nat-free. Se cumple entonces que $\text{máx}\{e_1, \ldots, e_n\} \in \Theta\left(\sum_{i=1}^{n} e_i\right)$. Nótese que esto es una consecuencia trivial, pues $\text{máx}\{e_1, \ldots, e_n\} \leq \sum_{i=1}^{n} e_i \leq n * max\{e_1, \ldots, e_n\}$. □

Propiedad 14 (Monotonía del Producto)**.** Sean e_1, e_2, e_3, e_4, e expresiones de coste nat-free. Entonces se cumple:

1. $e_1 \in \mathcal{O}(e_3) \wedge e_2 \in \mathcal{O}(e_4) \Rightarrow e_1 * e_2 \in \mathcal{O}(e_3 * e_4)$.
2. $e_1 \in \Theta(e_3) \wedge e_2 \in \Theta(e_4) \Rightarrow e_1 * e_2 \in \Theta(e_3 * e_4)$.

□

Propiedad 15 (Monotonía Asintótica de la Potencia sobre la Base)**.** Sean e_1, e_2 dos expresiones de coste nat-free. Siempre se cumple que:

1. Si $e_1 \in \mathcal{O}(e_2)$ entonces $\forall r \in \mathbb{R}^+$ sucede que $e_1^r \in \mathcal{O}(e_2^r)$.
2. Si $e_1 \in \Theta(e_2)$ entonces $\forall r \in \mathbb{R}^+$ se verifica que $e_1^r \in \Theta(e_2^r)$. □

Propiedad 16 (Monotonía Asintótica de la Potencia sobre el Exponente)**.** Para cada par $r, s \in \mathbb{R}^+$, $r, s > 0$, tales que $r \leq s$ se cumple que $e^r \in \mathcal{O}(e^s)$, siendo e cualquier expresión de coste nat-free. □

Propiedad 17 (Monotonía Asintótica del Logaritmo)**.** Si $e_1 \in \mathcal{O}(e_2)$ entonces $\log(e_1) \in \mathcal{O}(\log(e_2))$. □

Propiedad 18 (Dominio Polinómico - Logarítmico)**.** Para cada $r \in \mathbb{R}^+$, $r > 0$, se cumple que: $\log(e) \in \mathcal{O}(e^r)$. Por tanto, si $r, s \in \mathbb{R}^+$, $r, s > 0$, siempre se cumple que $\log(e)^s \in \mathcal{O}(e^r)$. □

Propiedad 19. Sean $e_1, ..., e_n, e, e'$ expresiones de coste nat-free y φ una restricción lineal tal que $\varphi \models e_i \geq 1$, $\varphi \models e_i \leq e$, $1 \leq i \leq n$. Entonces $\forall r, s_1, ..., s_n, t \in \mathbb{R}^+$ tal que $r > s_1 + ... + s_n$ se cumple que

$$\varphi \models \prod_{i=1}^{n} e_i^{s_i} * log^t(e') \in \mathcal{O}(e^r)$$

□

Demostración. Si descomponemos $e^r = e^{r'} * e^{s'}$, donde $s' = \sum_{i=1}^{n} s_i$ y $r' = r - s'$, obtenemos que $r' > 0$ y por tanto podemos aplicar la Propiedad 12 porque

- $\varphi \models \prod_{i=1}^{n} e_i^{s_i} \leq e^{s'}$.
- $\log(e')^t \in \mathcal{O}(e^{r'})$, aplicando la propiedad 18.

□

Propiedad 20 (Dominio Exponencial - Polinómico)**.** Para cualquier $r, s \in \mathbb{R}^+$, $s > 0$, se cumple que $e^r \in O(2^{s*e})$. □

Propiedad 21. Sean $e_1, ..., e_n, e, e'$ expresiones de coste nat-free y φ una restricción lineal tal que $\varphi \models 0 \leq e_i \leq e$. Entonces $\forall r, s_1, ..., s_n, t \in \mathbb{R}^+$ tales que $r > s_1 + ... + s_n$, se cumple que

$$\varphi \models 2^{\sum_{i=1}^{n} s_i * e_i} * e'^t \in O(2^{r*e})$$

□

Demostración. Primero descomponemos $2^{r*e} = 2^{r'*e} * 2^{s'*e}$, siendo $s' = \sum_{i=1}^{n} s_i$ y $r' = r - s' > 0$. Usando la Propiedad 12 nos basta ver que:

1. $\sum_{i=1}^{n} s_i * e_i \leq s' * e$.
2. $(e')^t \in \mathcal{O}(2^{r'*e})$ como consecuencia de la Propiedades 18 y 20.

□

4.3. Reducción de Expresiones de Coste

Para conseguir la Motivación de Legibilidad expuesto en el Capítulo. 1 nos interesa encontrar para cada orden el elemento de mínima sintaxis. En esta sección presentamos un mecanismo para reducir una expresión de coste a una forma asintóticamente equivalente y sin términos redundantes. Este algoritmo ya puede simplificar el resultado de un analizador en otro resultado asintóticamente equivalente.

Ejemplo 92 (Ejemplo intuitivo de subsunción asintótica)**.** La CExp del Ej. 89 está en el mismo orden de

$$13 * 3^{\mathsf{nat}(y)+1} * \mathsf{nat}(x-1)^3 \qquad \text{y} \qquad 23 * 3^{\mathsf{nat}(y)+1} * \mathsf{nat}(x)^3 + \mathsf{nat}(x)^{\frac{5}{2}}$$

pero cuando se trata de representar el orden asintótico, es preferible emplear la CExp más simple: $3^{\mathsf{nat}(y)} * \mathsf{nat}(x)^3$. □

Dada una expresión de coste nat-free $\tilde{e}$, describimos cómo simplificarla para obtener otra CExp nat-free $\tilde{e}'$ tal que $\tilde{e} \in \Theta(\tilde{e}')$. Si e es simplemente una constante (o expresión aritmética equivalente), ya sabemos que $\tilde{e} \in \mathcal{O}(1)$.

El caso más fácil de reducir son las expresiones constantes, que están en $\Theta(1)$. También hay algunas expresiones de coste sencillas que pueden simplificarse a forma asintótica con una reescritura que suprima paréntesis y constantes.

Ejemplo 93 (Simplificaciones asintóticas sencillas.)**.** Las CExp $10 * \mathsf{nat}(X+1)+14$ o $5*(\mathsf{nat}(X-1)+2)$ pertenecen a $\Theta(\mathsf{nat}(X))$. Podemos simplificarlas solo con suprimir constantes multiplicativas y aditivas. □

Pero en la mayoría de casos esto no basta. En ellos tenemos que manejar funciones compuestas de expresiones no lineales con subexpresiones exponenciales, polinómicas y logarítmicas de múltiples variables, las cuales están sometidas a diversas restricciones. Para mejorar la simplificación, resulta esencial eliminar subexpresiones redundantes, es decir cuyo aporte a la complejidad de la expresión es nulo. Tales subexpresiones corresponden a sumandos subsumidos.

Ejemplo 94 (Subsunción asintótica)**.** En $\mathsf{nat}(x)^2+\mathsf{nat}(x)$ el sumando $\mathsf{nat}(x)$ es redundante pues está subsumido por $\mathsf{nat}(x)^2$. □

Para encontrar términos subsumidos se requiere una comparación efectiva de expresiones de coste de distinta longitud y orden de complejidad.

4.3.1. La transformación τ

El primer paso es transformar $\tilde{e}$ eliminando las expresiones constantes y máx, como se describe en esta definición:

Definición 56 (Transformación τ)**.** Dada una expresión de coste nat-free $\tilde{e}$, $\tau(\tilde{e})$ denota la expresión de coste que se obtiene desde $\tilde{e}$

1. eliminando todas las constantes aditivas y multiplicativas y
2. reemplazando cada subexpresión $max(\{\tilde{e}_1, \ldots, \tilde{e}_m\})$ por otra subexpresión $(\tilde{e}_1 + \ldots + \tilde{e}_m)$. □

Ejemplo 95 (Transformación τ)**.** Para la expresión de coste nat-free del Ejemplo 89,

$$\tau(\tilde{e}) = A*(A^2*B^4 + 3^B*A^2) + \log(A)*2^B*\log(B)*C$$

es el resultado de aplicar τ. □

Lema 5. $\tilde{e} \in \Theta(\tau(\tilde{e}))$

Demostración. En el caso de las subexpresiones máx, es una consecuencia de la Propiedad 13. Para las demás operaciones, el resultado se deriva de la Propiedades 12 (suma), 1 (producto), 15 (potencias) y 17 (logaritmos). □

4.3.2. Forma normal

Una vez hemos aplicado τ, vamos a transformar la expresión a una *forma normal* que consiste en una suma de productos.

Definición 57 (Expresión nat-free Básica)**.** Una expresión de coste nat-free es básica si tiene alguna de estas formas:

- Una exponencial 2^{r*A} siendo $r \in \mathbb{R}^+$ el exponente.
- Un polinomio A^r, siendo $r \in \mathbb{R}^+$ el grado.
- Un logaritmo $\log(A)$. □

Nótese que sin pérdida de generalidad podemos suponer que todas las exponenciales tienen base 2 porque para cualquier $n \in \mathbb{N}$ con $n > 2$ se cumple $n^A = 2^{\log(n)*A}$.

Definición 58 (Forma Normal Asintótica de Expresiones nat-free)**.** Una expresión de coste nat-free *en forma normal* tiene la forma

$$\sum_{i=1}^{n}\prod_{j=1}^{m_i} b_{ij}$$

donde cada b_{ij} es una expresión de coste nat-free básica. □

Gracias a la propiedad conmutativa y asociativa de la suma y el producto de expresiones de coste, cada producto puede representarse como un multiconjunto M de factores (cada uno una expresión de coste nat-free básica) y cada suma como un multiconjunto de productos.

Proposición 5 (Normalización Asintótica de Expresiones de Coste)**.** Para toda CExp nat-free e existe una CExp e' en forma asintótica normal tal que $e \in \Theta(e')$ □

Demostración. Para normalizar una expresión de coste nat-free $\tau(\tilde{e})$ aplicaremos la propiedad distributiva del producto respecto a la suma para suprimir todos los paréntesis. □

Ejemplo 96 (Normalización Asintótica de Expresión de Coste)**.** La expresión

$$A^3{*}B^4{+}2^{\log(3){*}B}{*}A^3{+}\log(A){*}2^B{*}\log(B)*C$$

es la forma normal de la expresión del Ejemplo 95. □

4.3.3. Subsunción entre sumandos

Dada una expresión de coste nat-free en forma normal $\tilde{e} = \{M_1, \ldots, M_n\}$ y una restricción de contexto φ, queremos quitar de $\tilde{e}$ cualquier producto M_i que esté *asintóticamente subsumido* por otro producto M_j, esto es si $M_j \in \Theta(M_j + M_i)$, para lo que basta saber que $M_i \in \mathcal{O}(M_j)$.

Primero, definimos varios **patrones de subsunción asintótica** para los cuales resulta fácil verificar que una sola expresión de coste nat-free básica b subsume un producto completo. Estos patrones toman una CExp nat-free básica b con una sola variable A, y definimos cuándo b subsume asintóticamente un producto de CExp nat-free básicas. Para ello se observa el valor de las funciones auxiliares pow y deg.

Definición 59 (Funciones pow, deg)**.** Sea b una CExp básica nat-free. Se definen las funciones de exponente y grado asintótico $\mathsf{pow}, \mathsf{deg} :: CExp \to \mathbb{R}^+$ con estas ecuaciones:

$$\begin{array}{ccc} \mathsf{pow}(2^{r*A}) = r & \mathsf{pow}(A^r) = 0 & \mathsf{pow}(\log(A)) = 0 \\ \mathsf{deg}(A^r) = r & \mathsf{deg}(2^{r*A}) = \infty & \mathsf{deg}(\log(A)) = 0 \end{array}$$

Nótese que un exponencial tiene grado ∞. □

Lema 6 (Subsunción Asintótica Expresión-Producto)**.** Sean una expresión de coste nat-free básica b tal que $vars(b) = \{A\}$, un producto $M = b_1 * \cdots * b_m$ de m factores tales que $vars(b_i) = \{A_i\}$ y una restricción de contexto φ tal que $\varphi \models A \succeq A_i$. M está asintóticamente subsumida por b bajo φ, escrito $\varphi \models M \in \mathcal{O}(b)$ si se da uno de estos casos:

1. Exponencial:Si $b = 2^{r*A}$, entonces
   a) $r > \sum_{i=1}^{m} \mathsf{pow}(b_i)$; o
   b) $r \geq \sum_{i=1}^{m} \mathsf{pow}(b_i)$ y cada b_i tiene la forma $2^{r_i * A_i}$. Es decir, que M no contiene factores polinómicos o logarítmicos.
2. Polinómico: Si $b = A^r$, entonces
   a) $r > \Sigma_{i=1}^{m} \mathsf{deg}(b_i)$; o
   b) $r \geq \Sigma_{i=1}^{m} \mathsf{deg}(b_i)$ y cada b_i es de la forma $A_i^{r_i}$, es decir que M no contiene factores exponenciales ni logarítmicos.
3. Logarítmico: Si $b = \log(A)$ y M solo contiene un factor $b_1 = \log(A_1)$

□

Demostración. Siendo $M = b_1 * ... * b_n$, debemos probar que $M \in \mathcal{O}(b)$. Sean los números $p = \sum_{i=1}^{n} \mathsf{pow}(b_i)$ y $g = \sum_{i=1}^{m} \mathsf{deg}(b_i)$. Entonces podemos distinguir los siguientes tres casos:

1. Exponencial: Supongamos que $b = 2^{r*A}$, para algún $r \in \mathbb{R}^+$. Entonces:
   a) En $(1a)$ señalamos que si los exponentes son distintos, el de mayor exponente subsume al de menor exponente. Si $r > p$ la Propiedad 21 sostiene el resultado.
   b) En igualdad de exponentes es necesario que no haya más factores. Si $r \geq p$ y todos los b_i son de la forma $2^{r_i * A_i}$, siendo $r_i = \mathsf{pow}(b_i)$, podemos utilizar que

$$\varphi \models r * A \geq \sum_{i=1}^{n} r_i * A_i$$

   para inferir $b = 2^{r*A} \geq 2^{p*A}$.
2. Polinómico: Supongamos $b = A^r$ para algún $r \in \mathbb{R}^+$ y $p = 0$. Entonces:

a) $(2a)$ captura que si los grados son distintos, el de mayor grado es la cota superior aunque al de menor grado lo acompañen muchos factores logarítmicos. Esto se basa en la Propiedad 19.

b) $(2b)$ señala que una expresión polinómica solo subsume a otra con igual grado si ésta no incluye factores logarítmicos o exponenciales. Así, A no domina a $A_i * log(A_i)$. Solo tenemos que considerar que

$$b^r \geq \prod_{i=1}^{n} b_i^{g_i}$$

3. El caso Logarítmico expresa que un logaritmo solo está subsumido por otro logaritmo. Se deriva de la Propiedad 17.

□

Ejemplo 97 (Subsunción Asintótica Expresión - Producto)**.** Sea $b = A^3$, $M = \{\log(A), \log(B), C\}$, donde A, B y C corresponden respectivamente a $\mathsf{nat}(x)$, $\mathsf{nat}(y)$ y $\mathsf{nat}(x-y)$, y la restricción de contexto $\varphi = \{x \geq y, x \geq 0, y \geq 0\}$. M está asintóticamente subsumida por b puesto que $\varphi \models (A \succeq B) \wedge (A \succeq C)$, y se cumple la condición $2a$ del Lema 6. □

Mostramos ahora cómo inferir que dos productos M_1, M_2 de expresiones básicas cumplen $M_2 \in \mathcal{O}(M_1)$. La idea básica es buscar una factorización de M_2 que verifique que cada subproducto de M_2 está subsumido por un factor distinto de M_1.

Lema 7 (Subsunción Asintótica Producto - Producto)**.** Sean M_1, M_2 dos productos. Si existe una factorización de M_2 en k subproductos $S_1, \ldots, S_k$ con $k \leq n$, $k \leq m$ y hay k expresiones distintas nat-free básicas $b_1, \cdots, b_k$ que son factores de M_1, tales que cada subproducto S_i cumple $S_i \in \mathcal{O}(b_i)$ entonces $S_1 * ... * S_k \in \mathcal{O}(b_{f(1)} * ... * b_{f(k)})$ y por tanto $M_2 \in \mathcal{O}(M_1)$. □

Demostración. Es una extensión de la Propiedad 1. □

Ejemplo 98 (Subsunción asintótica producto-producto)**.** Sean los productos $M_1 = \{2^{\log(3)*B}, A^3\}$ y $M_2 = \{\log(A), 2^B, \log(B), C\}$ y la relación de contexto φ definida en el Ejemplo 97. Si tomamos $b_1 = 2^{\log(3)*A}$, $b_2 = A^3$ç y factorizamos M_2 en los subproductos $P_1 = \{2^B\}$, $P_2 = \{\log(A), \log(B), C\}$, entonces podemos ver que $P_1 \in \mathcal{O}(b_1)$ y $P_2 \in \mathcal{O}(b_2)$. Por tanto, aplicando el Lema 7, $M_2 \in \mathcal{O}(M_1)$.

Asimismo, para $M_2' = \{A^3, B^4\}$ podemos dividirlo en $P_1' = \{B^4\}$ y $P_2' = \{A^3\}$ de tal manera que $P_1' \in \mathcal{O}(b_1)$ y $P_2' \in \mathcal{O}(b_2)$, y por tanto también tenemos que $M_2' \in \mathcal{O}(M_1)$. □

4.3.4. Transformación `asymp`

Definición 60 (Transformación `asymp`)**.** Dada una expresión de coste e, la transformación `asymp` toma e y devuelve la expresión de coste que resulta tras

1. Obtener la forma nat-free,
2. aplicar la transformación τ,
3. normalizar a suma de productos,
4. suprimir todos los productos subsumidos,
5. y reemplazar cada variable nat por su expresión nat correspondiente. □

El siguiente teorema nos asegura que suprimir productos asintóticamente subsumidos no modifica el orden exacto de complejidad:

Teorema 1 (Corrección de `asymp`)**.** Dada una expresión de coste e y una restricción de contexto φ, se cumple que $\varphi \models e \in \Theta(\texttt{asymp}(e))$.

Demostración. Supongamos que e se descompone en $m_1 + \ldots + m_{n+k}$ y que los últimos k sumandos están subsumidos por los n primeros. Si definimos

$$e' = \texttt{asymp}(e) = \sum_{i=1}^{n} m_i \qquad\qquad e = e' + \sum_{i=1}^{k} m_{n+i}$$

donde cada m_{n+i} verifica $m_i \in \mathcal{O}(e')$, entonces podemos aplicar la Propiedad 12.3 y obtenemos que $e' + \sum_{i=1}^{k} m_{n+i} \in \Theta(e')$. □

Ejemplo 99 (Transformación `asymp`)**.** Considera la expresión de coste en forma normal del Ejemplo 96. Podemos eliminar el primer y tercer producto, que están subsumidos por el segundo como se explica en el Ejemplo 98. Entonces $\texttt{asymp}(e)$ sería

$$2^{\log(3)*\mathsf{nat}(y)} * \mathsf{nat}(x)^3 = 3^{\mathsf{nat}(y)} * \mathsf{nat}(x)^3$$

y se cumple que $e \in \Theta(\texttt{asymp}(e))$. □

4.4. Resolución Asintótica de *CRS*

Nuestro objetivo es construir un resolutor asintótico de *CRS*, que genere resultados asintóticos sin calcular primero los resultados no asintóticos. Para ello modificamos el proceso descrito en el Capítulo 3, para que maneje directamente funciones asintóticas en sus etapas intermedias y sus resultados sean cotas asintóticas.

4.4.1. Composicionalidad

Así como el algoritmo de resolución de cotas superiores se basa en el principio de composicionalidad (Propiedad 7), también debemos proporcionar un principio de composicionalidad para las cotas asintóticas. Es decir, si reeemplazamos una llamada externa a D por una cota superior asintótica D^+_{asymp} entonces toda cota asintótica de la relación modificada lo es también de la original.

Teorema 2 (Principio de Composicionalidad Asintótica)**.** Sea C una CR directamente recursiva con $k \geq 0$ llamadas a relaciones externas y n llamadas recursivas:

$$\langle C(\bar{x}) = e + \sum_{i=1}^{k} D_i(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$$

Sean $D^+_{i_{asymp}}(\bar{y}_i)$ la cota superior asintótica de $D_i(\bar{y}_i)$ y sea $C^+_{asymp}(\bar{x})$ la cota superior asintótica de la relación aislada

$$\langle C(\bar{x}) = e + \sum_{i=1}^{k} D^+_{i_{asymp}}(\bar{y}_i) + \sum_{j=1}^{n} C(\bar{z}_j), \varphi \rangle$$

Entonces, se cumple que $C^+(\bar{x}) \in O(C^+_{asymp}(\bar{x}))$ □

Demostración. Para probar $C^+(\overline{x}) \in O(C^+_{asymp}(\overline{x}))$ basta demostrar que

$$e + \sum_{i=1}^{k} D_i(\overline{y}_i) \in O\left(e + \sum_{i=1}^{k} D^+_{i_{asymp}}(\bar{y}_i)\right)$$

lo que puede hacerse por inducción sobre el número de llamadas externas k y por aplicación de la Propiedad 12. □

Hay que señalar que el proceso de sustituir una llamada por una cota superior asintótica puede introducir imprecisión.

Require: $\mathcal{S}$ está en forma directamente recursiva
1: **function** RESUELVE($\mathcal{S}$)
2: Tabla[CR,CExp] aubs = tablaVacía()
3: $lista = \mathcal{S}.daOrdenTopologico()$
4: **for** $C : lista$ **do**
5: $C_A^+ = C.resuelveCotaSuperiorAsintotica()$
6: $ubs.put(C, C^+)$
7: $\mathcal{S}.despliegaLlamadas(C, C_A^+)$
return ubs

Figura 4.1: Algoritmo de Resolución Asintótica de un *CRS* Directamente Recursivo

Ejemplo 100. Considere la expresión de coste

$$ub{=}\mathsf{nat}(a{-}b{+}1){*}2^{\mathsf{nat}(c)}{+}5$$

cuya forma asintótica es $\mathtt{asymp}(ub){=}\mathsf{nat}(a{-}b){*}2^{\mathsf{nat}(c)}$. Al expandir ub en un contexto donde $b{=}a{+}1$ daría como resultado 5 (pues $\mathsf{nat}(a{-}b{+}1){=}0$). Pero al expandir $\mathtt{asymp}(ub)$ en el mismo contexto el resultado sería $2^{\mathsf{nat}(c)}$ que es claramente menos preciso (de constante a exponencial). □

Intuitivamente, la pérdida de precisión se debe a que en el órden asintótico siempre se supone que las subexpresiones nat tienen valor muy grande, mientra que en los casos como el anterior siempre tienen un valor nulo. En la práctica es raro encontrarse con un *CRS* así, y por ese motivo no hemos estudiado aun cómo evitarlo, pero intuimos que solo haría falta detectar cuándo la expresión lineal de un nat está acotada en todos los contextos de llamada.

De cualquier modo, el principio de composicionalidad de las cotas asintóticas logra el mismo efecto que el de las no asintóticas: podemos resolver un *CRS* en forma directamente recursiva, si tratamos una a una sus CR siguiendo podemos resolver todas las CRs procesándolas en un orden topológico inverso del grafo de llamadas.

Al igual que para la resolución no asintótica, necesitamos un método que transforme un *CRS* a otro equivalente en forma directamente recursiva, para lo que podemos utilizar el algoritmo de Evaluación Parcial descrito en la Sección 3.3.

4.4.2. CR Asintóticas

El primer paso es transformar las CRs a forma asintótica antes de inferir cotas superiores. Así conseguimos simplificar el cálculo de los casos pésimos de las ecuaciones recursivas y de los casos base.

Definición 61 (Traducción a CR Asintótica)**.** Sea C una CR independiente. La traducción asintótica de C es la CR C_A que surge de convertir a una ecuación de C_A

$$\langle C_A(\bar{x}) = \mathtt{asymp}(e) + \sum_{i=1}^{k} C_A(\bar{y}_i), \varphi\rangle$$

cada ecuación de C de la forma $\langle C(\bar{x}) = e + \sum_{i=1}^{k} C(\bar{y}_i), \varphi\rangle$ □

Ejemplo 101 (CR asintótica)**.** Observe el siguiente CR:

$$\begin{array}{ll}
\langle C(a,b) = \underline{\mathsf{nat}(a+1)^2} & ,\{a{\geq}0, b{\geq}0\}\rangle \\
\langle C(a,b) = \underline{\mathsf{nat}(a{-}b){+}\log(\mathsf{nat}(a{-}b))}{+}C(a',b') & ,\{a{\geq}0, b{\geq}0, a'{=}a{-}2, b'=b+1\}\rangle \\
\langle C(a,b) = \underline{2^{\mathsf{nat}(a+b)} + 5 * \mathsf{nat}(a)^2}{+}C(a',b') & ,\{a{\geq}0, b{\geq}0, a'{=}a{+}1, b'=b-1\}\rangle
\end{array}$$

Al reemplazar las expresiones subrayadas por sus correspondientes expresiones `asymp`, tal y como se explica en el Teorema 1, obtenemos la siguiente relación asintótica:

$$\begin{array}{lcll}
\langle C_A(a,b) & = & \mathsf{nat}(a)^2 & , \{a{\geq}0, b{\geq}0\}\rangle \\
\langle C_A(a,b) & = & \mathsf{nat}(a{-}b){+}C_A(a',b') & , \{a{\geq}0, b{\geq}0, a'{=}a{-}2, b'{=}b{+}1\}\rangle \\
\langle C_A(a,b) & = & 2^{\mathsf{nat}(a+b)}{+}C_A(a',b') & , \{a{\geq}0, b{\geq}0, a'{=}a{+}1, b'{=}b{-}1\}\rangle
\end{array}$$

La forma asintótica canónica no solamente es más pequeña y por tanto ocupa menos, sino que también tiene menos subexpresiones `nat` y por tanto la maximización de costes locales es más eficiente. □

Las relaciones de tamaño

Si bien es posible transformar e, no se debe modificar en absoluto la relación de tamaño φ de cada ecuación de coste. De lo contrario, obtendríamos resultados incorrectos. En el cálculo de cotas superiores, φ se usa para calcular las funciones de rango y los invariantes de la CR, y si se modifican podríamos perder la precisión o incluso la corrección de nuestros análisis. Para explicarlo, veamos algunos ejemplos:

Ejemplo 102 (Incorrección de transformar la relación de tamaño.)**.** La CR anterior admite como función de rango $f(a,b){=}\mathsf{nat}(2a{+}3b{+}1)$. Pero si suprimimos las constantes de la relación de tamaño, cambiando $a'{=}a{-}2$ por $a'{=}a$, la relación que obtenemos es no terminante (y por tanto no admite ninguna función de rango). □

Por otro lado, eliminar las constantes en las restricciones que no están directamente relacionadas a la función de rango también puede hacer que calculemos invariantes incorrectos.

Ejemplo 103. Si cambiamos $n'{=}n{+}1$ por $n'{=}n$ en esta ecuación:

$$\langle C(m,n) = \mathsf{nat}(n) + C(m',n')\ ,\ \{m{>}0, m'{<}m, n'{=}n{+}1\}\rangle$$

Obtendremos un invariante que asegura que el valor de n siempre es el mismo que el valor inicial n_0. Este invariante nos llevaría a calcular $\mathsf{nat}(m_0){*}\mathsf{nat}(n_0)$ como cota superior, lo que es incorrecto. □

4.4.3. Cotas superiores Asintóticas

Una vez que la CR aislada es pasada a forma asintótica, procedemos a calcular la cota superior para ella al igual que para la no asintótica, y entonces transformamos el resultado.

Definición 62 (Cota superior asintótica C^+_{asymp})**.** Sean

- $C_A(\bar{x})$ una relación de coste asintótica y
- $C^+_A(\bar{x})$ su cota superior, ya haya sido calculada mediante la fórmula de acumulación de evaluaciones (Proposición 3) o la de acumulación por niveles (Proposición 4).

Entonces la expresión de coste C^+_{asymp}

$$C^+_{asymp}(\bar{x}) = \mathtt{asymp}(C^+_A(\bar{x}))$$

es una cota superior asintótica de C.

Observe que calculamos $C^+_A(\bar{x})$ de manera no asintótica, esto es sin aplicar `asymp` en cada subexpresión $\mathsf{l_b}$, $\mathsf{l_r}$, *worst* de las fórmulas de acumulación de evaluaciones o por niveles. Podríamos aplicar `asymp` a cada elemento de cada una de la fórmula que usáramos, pero esto apenas sí tiene un efecto notable.

Ejemplo 104 (Resolución Asintótica de *CRS*)**.** Considere la segunda CR del Ejemplo 101.

- El analizador calcula el invariante $\mathcal{I} = \{0{\leq}a{\leq}a_0, 0{\leq}b{\leq}b_0, a{\geq}0, b{\geq}0\}$
- Con éste, maximizamos $\mathsf{nat}(a)^2$ a $\mathsf{nat}(a_0)^2$, $\mathsf{nat}(a{-}b)$ a $\mathsf{nat}(a_0)$ (el valor máximo se da cuando $b = 0$), y $2^{\mathsf{nat}(a+b)}$ en $2^{\mathsf{nat}(a_0+b_0)}$.
- El número de iteraciones desus bucles es $\mathsf{nat}(2a_0{+}3b_0{+}1)$ (Ejemplo 102).

Al aplicar la fórmula de la Proposición 3, obtenemos la cota superior

$$C^+_A(a,b) = \mathsf{nat}(2a{+}3b{+}1) * \text{máx}(\{\mathsf{nat}(a), 2^{\mathsf{nat}(a+b)}\}) + \mathsf{nat}(a)^2$$

y al aplicar `asymp` a se obtiene $C^+_{asymp}(a,b) = 2^{\mathsf{nat}(a+b)} * \mathsf{nat}(2a+3b)$. □

4.5. Experimentos en COSTA

Ya hemos visto el algoritmo de transformación `asymp` que transforma expresiones de coste a forma asintótica. También hemos visto cómo integrar esta transformación en el proceso de resolución de un *CRS* para que genere directamente resultados en forma cerrada asintótica.

Ahora debemos comprobar si estos métodos consiguen, efectivamente, lograr las motivaciones que mencionamos en la Subsección 1.5.2 y que recordamos a continuación

- Queremos que los resultados asintóticos sean más legibles para que el programador pueda interpretarlos con mayor facilidad.
- Al usar un proceso de resolución asintótica se debe mejorar la escalabilidad del cálculo, así que el resolutor asintótico debería tardar menos que el no asintótico para un *CRS* grande.

Esta sección muestra en qué prototipo se han desarrollado estos algoritmos, con qué programas de entrada los hemos probado y si los resultados obtenidos cumplen nuestro desideratum.

4.5.1. Implementación: COSTA y PUBS

COSTA ***(Cost and Termination Analyzer)*** es un analizador del coste de ejecución y de la terminación de los programas escritos en Código byte de Java *(Java Bytecode JBC)*. COSTA toma como entradas un programa en JBC y un modelo de coste, que registra los valores para el recurso que nos interesa, y genera un *CRS* que captura los costes en ese programa.

Integra el subsistema PUBS ***(Practical Upper Upper Bound Solver)*** que resuelve los *CRS* y obtiene resultados en forma cerrada asintóticos y no asintóticos, que actualmente son solo cotas superiores. Utiliza PPL *(Parma Polyhedra Library)*, desarrollada por R. Bagnara et al. [12], para manipular las restricciones lineales. Actualmente, es posible probar su funcionamiento a través de la interfaz web disponible desde la página web del proyecto COSTA [1].

COSTA es un sistema implementado en Prolog y disponible para varias plataformas. Está planeada su próxima distribución bajo la *GNU General Public License*.

[1] `http://costa.ls.fi.upm.es`

4.5.2. Resultados de Legibilidad

Hay varios autores que han construido resolutores no asintóticos de relaciones de coste, como Braberman et al. [18], Gulwani [29], Hermenegildo et al. [44]), Chin et al. [22] y COSTA [7]. La transformación `asymp` puede usarse como filtro *back-end* a la salida de esos resolutores, como muestra la Figura 4.2. `asymp` acepta y procesa expresiones de coste de cualquier origen siempre que sigan la gramática de expresiones de coste de la Definición 5. No importa si éstas representan una cota superior o inferior de coste. Tampoco importa cómo se haya implementado el analizador que generó esas expresiones, o si este sigue la arquitectura de Wegbreit [57].

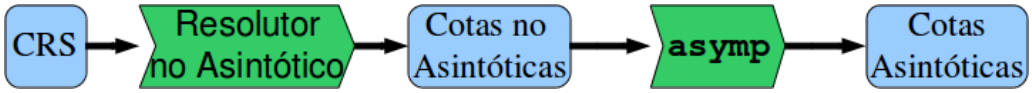

Figura 4.2: Uso de `asymp` como filtro

Como benchmarks hemos empleado los *CRS* obtenidos al analizar los programas de la suite JOlden [34] con el modelo del consumo de memoria. Esta suite fue usada por primera vez por Chin et al. en un artículo sobre verificación del consumo de memoria [21]. Desde entonces se está convirtiendo en un estándar para evaluar analizadores de consumo de memoria, y en el caso de COSTA ha sido empleada por Gómez, Genaim y Albert en [18, 9].

En los experimentos hemos acoplado la transformación `asymp` a la salida de PUBS para que procese los *CRS* obtenidos en esos artículos. En esos experimentos hemos comprobado que la transformación `asymp` convierte las cotas a forma asintótica en un tiempo prácticamente inapreciable (en ningún caso supera la diezmilésima de segundo).

Ejemplo 105 (Legibilidad)**.** En el benchmark `mst`, COSTA obtiene una cota superior de

$$16{*}\mathsf{nat}(a{+}1){*}\mathsf{nat}(a{+}1){+}4{*}\mathsf{nat}(a{+}1){*}\mathsf{nat}(\frac{1}{4}a){+}32{*}\mathsf{nat}(a{+}1){+}4$$

y nuestra transformación lo reduce a $\mathsf{nat}(a)^2$. En el benchmark `em3d`, la cota superior no asintótica es

$$8{*}\mathsf{nat}(d{-}1){*}\mathsf{nat}(b){+}8{*}\mathsf{nat}(d){+}8{*}\mathsf{nat}(b){+}56{*}\mathsf{nat}(d{-}1){+}16{*}\mathsf{nat}(c) + 73$$

y nuestra tranformación lo reduce a $\mathsf{nat}(d){*}\mathsf{nat}(b){+}\mathsf{nat}(c)$. RIL. □

Hay otros ejemplos que pueden probarse en la interfaz web de COSTA, accesible a través de la página web anteriormente mencionada.

4.5.3. Experimentos de Escalabilidad

En esta Subsección se prueba que un resolutor asintótico de coste que no construye primero los resultados no asintóticos es más escalable que uno no asintótico. El esquema de un resolutor así se muestra en la Figura 4.3, Con

Figura 4.3: Resolutor Asintótico de *CRS*

ese fin estudiamos experimentalmente cómo varía el tamaño de los resultados intermedios y de los tiempos de cálculo cuando se usan como datos de entrada varios *CRS* de distinto tamaño (medido en número de ecuaciones).

Benchmarks: PUBS

Para ello, hemos usado la suite de benchmarks diseñada por Albert et al. [10], que se muestran en la Tabla 4.1. Esta suite consiste en una serie de sencillos programas Java (ninguno excede las 50 LOC sin comentarios) que tratan problemas clásicos de Informática como copiar los datos de un array (Polynomial), el cálculo del logaritmo de un número (DivByTwo), el cálculo del Factorial, los algoritmos divide y vencerás de ordenación como MergeSort, la multiplicación de matrices MatMult, la fórmula (exponencial) del enésimo número de Fibonacci. El código fuente de estos programas está disponible a través de la página web de PUBS.

Esta suite tiene un gran interés porque con todos esos problemas se cubren los órdenes de complejidad más importantes. Polynomial tiene coste en $\Theta(1)$, DivByTwo es un ejemplar de $\Theta(log(n))$, hay un programa de orden lineal $\Theta(n)$ como Factorial, MergeSort tiene coste pésimo cuasilineal $\mathcal{O}(n \times log(n))$ e incluso varios exponenciales. Por tanto, esta suite sirve para probar que, en efecto, las CR pueden representar funciones de cualquier clase de complejidad (Subsección1.3.2).

Para emplearlos como benchmarks de escalabilidad, se ha realizado la misma operación que en [10]: se toman los *CRS* que COSTA genera para cada programa, en el orden en que aparecen en la Tabla 4.1, y se modifica cada *CRS* introduciendo en el caso recursivo de la CR principal una llamada a la CR principal del *CRS* del siguiente (inmediatamente inferior) benchmark.

Por tanto, cada fila de la Tabla corresponde al *CRS* del programa de la columna Bench al que se han añadido todas las CR de los *CRS* de los

Bench.	$\mathbf{T}_{ub}$	$\mathbf{T}_{aub}$	$\mathbf{Size}_{ub}$	$\mathbf{Size}_{aub}$	**#Eq**	$\frac{\mathbf{Size}_{ub}}{\mathbf{\#Eq}}$	$\frac{\mathbf{Size}_{aub}}{\mathbf{\#Eq}}$	$\frac{\mathbf{Size}_{ub}}{\mathbf{Size}_{aub}}$
BST	0	0	23	4	31	0,14	0,13	5,75
Fibonacci	0	0	47	9	39	1,21	0,23	5,22
Hanoi	0	0	67	14	48	1,39	0,29	4,78
MatMult	0	0	152	38	67	2,27	0,56	4,00
Delete	0	4	320	65	100	3,20	0,65	4,92
FactSum	4	4	717	95	117	6,12	0,81	7,54
SelectOrd	0	4	1447	155	136	10,63	1,14	9,33
ListInter	4	16	3804	257	173	21,98	1,48	14,80
EvenDigits	4	20	7631	400	191	39,95	2,09	19,07
Cons	12	32	15268	585	214	71,34	2,73	26,09
Power	24	40	24265	588	223	108,81	2,63	41,26
MergeList	96	60	48536	828	245	198,10	3,37	58,61
ListRev	140	76	48545	829	254	191,12	3,26	58,55
Incr	×	112	×	1126	282	×	3,99	×
Concat	×	164	×	1538	296	×	5,19	×
ArrayRev	×	232	×	2127	305	×	6,97	×
Factorial	×	284	×	2130	314	×	6,78	×
DivByTwo	×	328	×	2135	323	×	6,60	×
Polynomial	×	436	×	2971	346	×	8,58	×
MergeSort	×	440	×	3234	385	×	8,40	×

Cuadro 4.1: Escalabilidad de Expresiones de Coste Asintóticas

ejemplos inferiores más una llamada a la CR principal del inmediatamente inferior. La Columna **#Eq** muestra el número de ecuaciones que tiene el *CRS* así obtenido. Si leemos la columna descendentemente, se puede ver que cuando analizamos BST tenemos 31 Ecuaciones. Entonces, para Fibonacci, el número de ecuaciones es 39, es decir las 8 ecuaciones del *CRS* del programa Fibonacci más las 31 que se habían acumulado para BST. De esta manera en cada benchmark se añaden de manera progresiva las ecuaciones del *CRS* del programa de esa fila, hasta llegar a la última fila en la que se trata de resolver un *CRS* con 385 ecuaciones conectadas con que contiene al menos 20 bucles anidados.

Tiempos

En esta sección estudiamos cómo aumenta el relación de tamaño de las expresiones de coste, asintóticas y no asintóticas, cuando se analizan CRs mayores.

Las columnas $\mathbf{T}_{aub}$ y $\mathbf{T}_{ub}$ muestran cuánto tiempo tardan los resolutores asintótico y no asintótico en resolver cada *CRS* de prueba, medido en milésimas de segundo. El gráfico asociado puede verse en la Figura 4.4

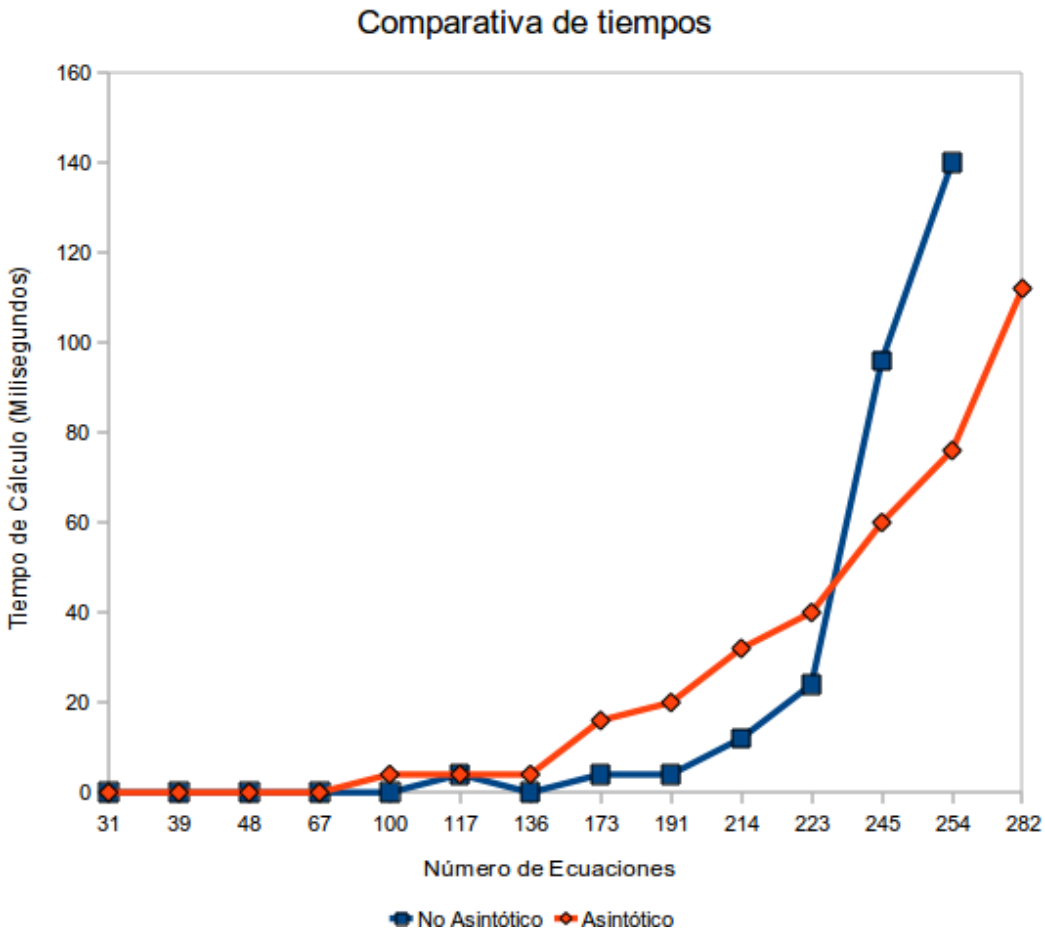

Figura 4.4: Comparativa de Tiempos de Cálculo.

En estas medidas se han sustraido el tiempo que tardan las operaciones que en ambos procesos son iguales, tales como la evaluación parcial (Sección 3.3), el cálculo de invariantes (Subsección 3.5.1) y el de funciones de rango (Sección 3.4). Las celdas en las cuales se ha escrito un aspa $\times$ corresponden a los casos en que el cálculo en PUBS falló. En algunos casos significa

que se desbordó la memoria del intérprete Prolog subyaciente, y en otros que el cálculo no terminaba en un tiempo razonable (como cinco minutos). Ello se debía a que no terminaba antes de cierto límite de tiempo o porque fallaba la plataforma Prolog.

Podemos observar que del benchmark BST a EvenDigits, que son los benchmarks más sencillos y con menos ecuaciones, los tiempos de cálculo apenas sí son apreciables. El punto interesante es que cuando las expresiones de coste empiezan a ser considerablemente largas, $\mathbf{T}_{ub}$ crece significativamente mientras que $\mathbf{T}_{aub}$ permanece pequeño. Esto se debe a la diferencia de tamaños de las expresiones que se manejan, como describimos más abajo. Las celdas que contienen "×" indican que COSTA no calculó una cota superior no asintótica.

Tamaño de las expresiones

Como en todos los programas, el coste de ejecución del resolutor depende de las expresiones y fórmulas que maneja. Al igual que en el análisis de programas (Subsección 1.1.3), nos conviene abstraer los datos del programa (las expresiones de coste) con alguna medida de tamaño.

La métrica que vamos a utilizar establece que el tamaño de una expresión de coste es, intuitivamente, el número de nodos de su árbol sintáctico.

Definición 63 (Tamaño Sintáctico de una Expresión de Coste)**.** Definimos la función de tamaño (size) de expresiones de coste $Size : CExp \mapsto \mathbb{N}$ con las siguientes ecuaciones

- El tamaño de una expresión lineal l es el número de variables más el número de coeficientes $c \neq 1$ de l.
- Si la expresión de coste es un número real $r \in \mathbb{R}^+$, su tamaño es 1.
- El tamaño de una expresión $\mathsf{nat}(l)$ es el tamaño de la expresión lineal más una unidad.
- El tamaño de una expresión $log_n(\mathsf{nat}(l)+1)$ o $n^{\mathsf{nat}(l)}$ es 2 más el tamaño de la expresión nat.
- El tamaño de una operación de suma, producto o máximo es 1 más la suma de los tamaños de sus operandos. □

Ejemplo 106 (Tamaño de una expresión de coste)**.** La expresión de coste $nat(x + 2 * y - 1) - 1$ tiene tamaño 7. □

Las columnas $\mathbf{Size}_{aub}$ y $\mathbf{Size}_{ub}$ dicen, para cada *benchmark*, cuánto miden las cotas superiores que calculan, respectivamente, el resolutor asintótico y el y no asintótico. El gráfico de la Figura 4.5 muestra cómo aumenta el tamaño de las expresiones según el tamaño del *CRS* de entrada.

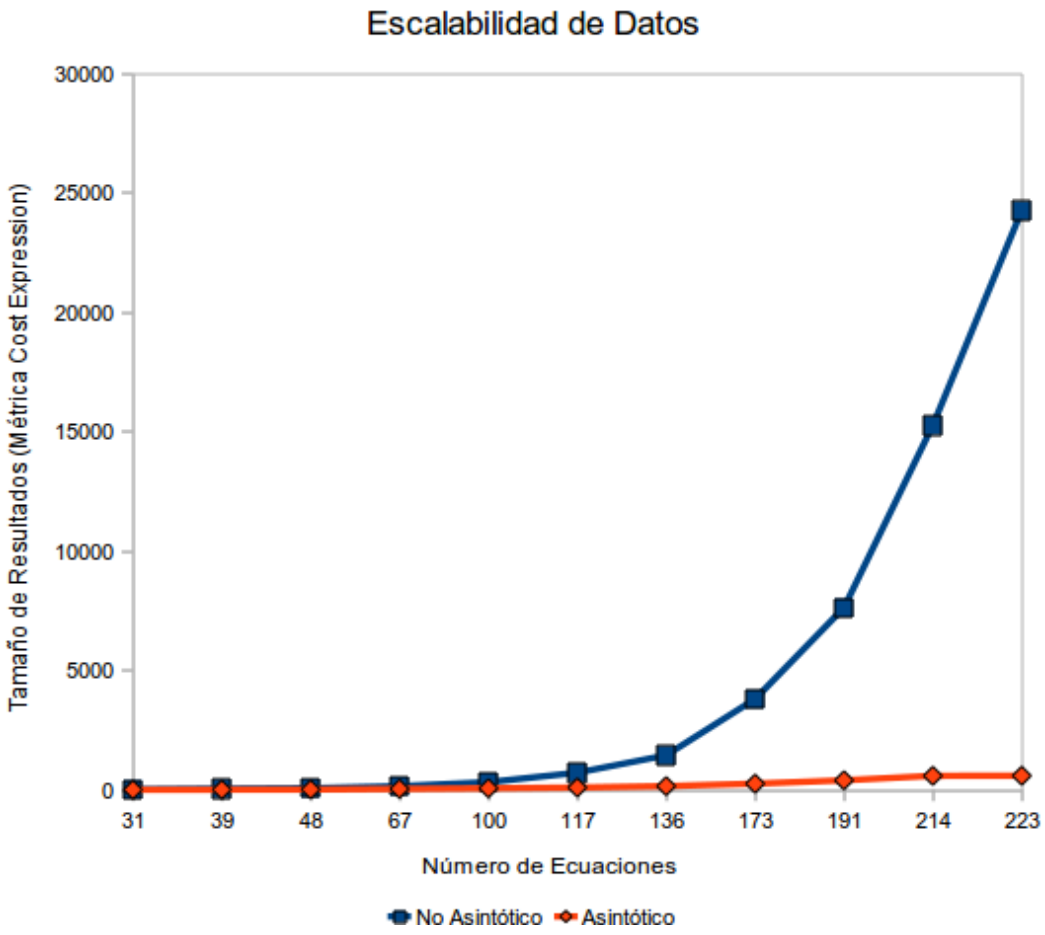

Figura 4.5: Comparativa Tamaños de los Resultados.

Para los benchmarks más sencillos, podemos ver que las expresiones de coste son bastante pequeñas, y empiezan a aumentar a partir del SelectOrd. A partir del benchmark SelectOrd empiezan a ser más grandes, pero mucho más las cotas superiores no asintóticas que las asintóticas. Se puede observar una correlación entre el tiempo de cálculo y el tamaño de los datos.

Ratios tamaño de expresión- número ecuaciones

Las columnas $\frac{\mathbf{Size}_{aub}}{\#\mathbf{Eq}}$ y $\frac{\mathbf{Size}_{ub}}{\#\mathbf{Eq}}$ muestran las ratios tamaño- número de ecuaciones para los resultados asintóticos y no asintóticos. La comparativa gráfica de esas ratios se muestra en el gráfico de la Figura 4.6. Lo más relevante es que mientras que esta ratio crece de manera exponencial para cotas superiores no asintóticas, crece mucho más lentamente en las asintóticas. $\frac{\mathbf{Size}_{aub}}{\#\mathbf{Eq}}$. Creemos que esto demuestra cómo este enfoque es escalable, aunque la implementación actual sea prototípica y preliminar.

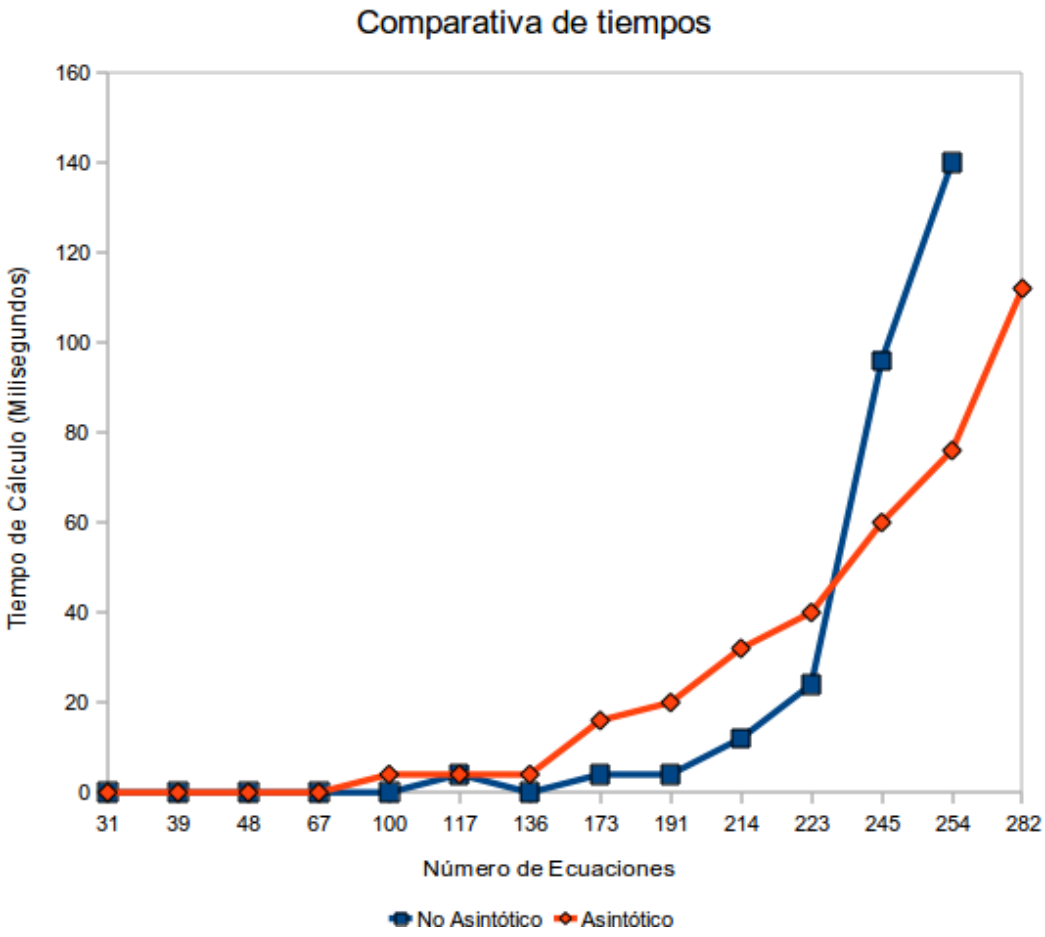

Figura 4.6: Comparativa de Ratios Tamaño de Expresión - Ecuaciones

Capítulo 5

Conclusiones

En este Trabajo de Fin de Máster:

- Revisamos la sintaxis y semántica de los sistemas de relaciones de coste y definimos los conceptos con los cuales describir sus propiedades más caracerísticas: el indeterminismo, la dependencia entre CR y la recursividad.
- Describimos intuitivamente el método desarrollado en [10] para resolver relaciones de coste.
- Adaptamos las nociones de órden asintótico $\mathcal{O}$ y equivalencia asintótica Θ para comparar expresiones de coste con múltiples variables y con restricciones asociadas.
- Desarrollar un procedimiento para simplificar una expresión de coste a forma asintótica.
- Construimos un resolutor asintótico de relaciones de coste que calcula cotas asintóticas sin obtener previamente las no asintóticas.

La complejidad asintótica sirve para describir la escalabilidad del coste de un programa. Si bien hasta ahora su análisis era efectuado manualmente, los resultados de este proyecto permiten realizarlo de manera automática.

Este proyecto proporciona el primer algoritmo que simplifica funciones de coste a su forma asintótica mínima, más legible y asequible. Asimismo, muestra cómo construir un resolutor asintótico y demuestra que éste tiene mejor rendimiento y escalabilidad que el no asintótico. Ambos métodos constituyen el primer enfoque genérico y automático de resolución asintótica de relaciones de coste, y es aplicable a una amplia clase de programas que cubren las principales necesidades de la industria software.

Este resolutor, unido al analizador COSTA, proporciona los fundamentos para construir una plataforma de desarrollo de programas JAVA capaz de integrar la verificación de aserciones de complejidad asintótica. La existencia de un *plugin* para el IDE ECLIPSE permite su uso con fines académicos, previo paso para su uso industrial.

5.1. Trabajo Futuro

El Análisis de Coste es un campo amplio en el que existen muchas metas que alcanzar.

Cotas inferiores: El método descrito en el Capítulo 3 sirve para obtener cotas **superiores** de relaciones de coste. Pero en algunas aplicaciones, como el equilibrado de carga (Sección 1.2), se necesitan cotas inferiores. Sería posible calcular éstas si se reorienta el método de resolución.

Análisis modular: Los sistemas de relaciones de coste (Sección 2.3) están concebidos como entidades monolíticas. Esto se adecúa al análisis de programas de mediano tamaño, pero no al del software industrial que suele ser construido ensamblando componentes reutilizables. Si se adapta la definición de *CRS* para convertirlos en módulos interconectados por interfaces, sería posible implementar un análisis estático modular de coste que analizare separadamente cada componente y reutilizare los resultados de los análisis.

Código Certificado: El empleo de las cotas superiores como certificados de código móvil permitiría construir una infraestructura de Código Certificado *(Proof-Carrying Code)*. Para ello sería preciso desarrollar e integrar en las plataformas móviles un verificador de resultados de coste.

Bibliografía

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] E. Albert, P. Arenas, D. Alonso, S. Genaim, and G. Puebla. Asymptotic Resource Usage Bounds. In Zhenjiang Hu, editor, *The Seventh Asian Symposium on Programming Languages and Systems (APLAS'09)*, Lecture Notes in Computer Science. Springer, 2009.

[3] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In Gilles Barthe and Frank de Boer, editors, *Proceedings of the IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 5051 of *Lecture Notes in Computer Science*, pages 2–18, Oslo, Norway, June 2008. Springer-Verlag, Berlin.

[4] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramirez, G. Román, and D. Zanardini. Termination and Cost Analysis with COSTA and its User Interfaces. In *Spanish Conference on Programming and Computer Languages (PROLE'09)*, ENTCS. Elsevier, September 2009. to appear.

[5] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Cost Relation Systems: a Language–Independent Target Language for Cost Analysis. In *Spanish Conference on Programming and Computer Languages (PROLE'08)*, volume 17615 of *ENTCS*. Elsevier, October 2008.

[6] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Applications of Static Slicing in Cost Analysis of Java Bytecode. In *3rd International Workshop on Programming Language Interference and Dependence (PLID'07)*, August 2007.

[7] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European*

Symposium on Programming, ESOP'07, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[8] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Post-proceedings of Formal Methods for Components and Objects (FMCO'07)*, number 5382 in LNCS, pages 113–133. Springer-Verlag, October 2008.

[9] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *ISMM'09: Proceedings of the 8th international symposium on Memory management*, New York, NY, USA, June 2009. ACM Press.

[10] Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 15-17, 2008, Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer-Verlag, July 2008.

[11] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proc. of Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 1–27. Springer, 2005.

[12] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[13] R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. Technical report, 2005. `arXiv:cs/0512056` available from `http://arxiv.org/`.

[14] Amir M. Ben-Amram. Size-change termination with difference constraints. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[15] Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In *Logic and Theory of Algorithms, 4th Conference on Computability in*

Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings, volume 5028 of *Lecture Notes in Computer Science*, pages 67–76. Springer, 2008.

[16] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.

[17] G. Bonfante, J-Y. Marion, and J-Y. Moyen. Quasi-interpretations and small space bounds. In J. Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2005.

[18] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *ISMM*. ACM Press, 2008.

[19] M. Braverman. Termination of Integer Linear Programs. In *Computer Aided Verification (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 372–385. Springer, 2006.

[20] A. Chander, D. Espinosa, N. Islam, P. Lee, and G.Ñecula. Enforcing resource bounds via static verification of dynamic checks. In *ESOP'05*, volume 3444 of *LNCS*. Springer, 2005.

[21] W.-N. Chin, H. H. Nguyen, S. Qin, and M. C. Rinard. Memory Usage Verification for OO Programs. In *Proc. of SAS'05*, volume 3672 of *LNCS*, pages 70–86, 2005.

[22] W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *ISMM*. ACM Press, 2008.

[23] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[24] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–97. ACM Press, 1978.

[25] Stephen-John Craig and Michael Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 23–34, New York, NY, USA, 2005. ACM Press.

[26] K. Crary and S. Weirich. Resource Bound Certification. In *POPL'00*, pages 184–198. ACM, 2000.

[27] S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[28] G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 2002.

[29] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.

[30] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *PPDP*. ACM Press, 2005.

[31] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[32] T. Hickey and J. Cohen. Automating program analysis. *J. ACM*, 35(1), 1988.

[33] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, 2003.

[34] JOlden Suite Collection. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.

[35] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.

[36] J. Komorovski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta Programming in Logic, Proceedings of META'92*, volume 649 of *LNCS*, pages 49–69. Springer-Verlag, 1992.

[37] Lars Kristiansen and Neil D. Jones. The flow of data and the complexity of algorithms. In S. Barry Cooper, Benedikt Löwe, and Leen Torenvliet, editors, *CiE*, volume 3526 of *Lecture Notes in Computer Science*, pages 263–274. Springer, 2005.

[38] D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[39] M. Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 26(3):413 – 463, May 2004.

[40] M. Leuschel and M. Bruynooghe. Logic Program Specialisation through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[41] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3–4):217–242, 1991.

[42] Beatrice Luca, Stefan Andrei, Hugh Anderson, and Siau-Cheng Khoo. Program transformation by solving recurrences. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 121–129, New York, NY, USA, 2006. ACM.

[43] J-Y. Marion and R. Péchoux. Resource analysis by sup-interpretation. In M. Hagiya and P. Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 163–176. Springer, 2006.

[44] J.Ñavas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2009.

[45] J.Ñavas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.

[46] G. Necula. Proof-Carrying Code. In *Proc. of ACM Symposium on Principles of programming languages (POPL)*, pages 106–119. ACM Press, 1997.

[47] K-H. Niggl and H. Wunderlich. Certifying Polynomial Time and Linear/Polynomial Space for Imperative Programs. *SIAM J. Comput.*, 35(5):1122–1147, 2006.

[48] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.

[49] G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 261–271. ACM Press, July 2006.

[50] M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.

[51] M. Rosendahl. Simple driving techniques. In T. Mogensen, D. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 404–419. Springer, 2002.

[52] D. Sands. Complexity Analysis for a Lazy Higher-Order Language. In *ESOP'00*, volume 432 of *LNCS*, pages 361–376. Springer, 1990.

[53] D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.

[54] Fausto Spoto, Patricia M. Hill, and Etienne Payet. Path-length analysis of object-oriented programs. In *Proc. International Workshop on Emerging Applications of Abstract Interpretation (EAAI)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[55] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[56] P. Wadler. Strictness analysis aids time analysis. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 119–132. ACM Press, 1988.

[57] B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.

[58] H. S. Wilf. *Algorithms and Complexity.* A.K. Peters Ltd, 2002.

Índice alfabético

Índice de figuras